SAMS





3D游戏编程大师技巧

TRICKS OF THE 3D GAME PROGRAMMING GURUS

ADVANCED 3D GRAPHICS AND RASTERIZATION

附光盘

[美] André LaMothe 著
李祥瑞 陈 武 译

人民邮电出版社
POSTS & TELECOM PRESS

全面覆盖软件3D光栅化、算法及实现的高级技术



SAMS

3D游戏编程大师技巧

高级3D图形和光栅化技术

本书是著名游戏编程图书作者André LaMothe撰写的《游戏编程大师技巧》的第二卷，本卷中对基于软件的3D光栅化、算法和实现进行了全面、深入的讲述。

- 本书中开发了一套先进的实时软件引擎；
- 手把手教读者从像素绘制和线框3D开始，一直到创建完整的16位z缓存3D引擎；
- 学习高级优化和编码技术；
- 探讨1/z缓存、mip-mapping、alpha混合、纹理滤波、空间划分等高级技术。











光盘内容

- 流行游戏开发工具的评估版
- DirectX 9.0 SDK
- 书中的全部源代码
- 丰富的多媒体素材
- 本书附录内容的电子文档
- 宝贵的3D图形学文章资料

ISBN 7-115-13371-9

9 787115 133717 >

人民邮电出版社网址 www.ptpress.com.cn

ISBN7-115-13371-9/TP·4645
定价：118.00元(附光盘)

3D 游戏编程大师技巧

[美] André LaMothe 著

李祥瑞 陈武 译

人民邮电出版

图书在版编目（CIP）数据

3D 游戏编程大师技巧 /（美）拉莫泽（Lamothe, A.）著；李祥瑞，陈武译.
—北京：人民邮电出版社，2005.6

ISBN 7-115-13371-9

Ⅰ. 3… Ⅱ. ①拉…②李…③陈… Ⅲ. 游戏—应用程序—程序设计—教材 Ⅳ. G899

中国版本图书馆 CIP 数据核字（2005）第 026252 号

版 权 声 明

André LaMothe: Tricks of the 3D Game Programming Gurus: Advanced 3D Graphics and Rasterization (ISBN: 0672318350)
Copyright © 2003 by Sams Publishing
All rights reserved.

本书中文简体字版由美国 Sams Publishing 出版公司授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

3D 游戏编程大师技巧

◆ 著 [美] André LaMothe
译 李祥瑞 陈 武
责任编辑 李 岚

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn

读者热线 010-67132705
北京鸿佳印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092 1/16
印张：58.75
字数：1 921 千字 2005 年 6 月第 1 版
印数：1–3 500 册 2005 年 6 月北京第 1 次印刷

著作权合同登记号 图字：01-2003-6127 号
ISBN 7-115-13371-9/TP • 4645

定价：118.00 元（附光盘）

本书如有印装质量问题，请与本社联系 电话：（010）67129223

内容提要

本书是游戏编程畅销书作者 André LaMothe 的扛鼎之作，从游戏编程和软件引擎的角度深入探讨了 3D 图形学的各个重要主题。全书共分 5 部分，包括 16 章的内容。第 1～3 章简要地介绍了 Windows 和 DirectX 编程，创建了一个 Windows 应用程序模板，让读者能够将精力放在游戏逻辑和图形实现中，而不用考虑 Windows 和 DirectX 方面的琐事；第 4～5 章简要地介绍了一些数学知识并实现了一个数学库，供以后编写演示程序时使用；第 6 章概述了 3D 图形学，让读者对本书将介绍的内容有大致的了解；第 7～11 章分别介绍了光照、明暗处理、仿射纹理映射、3D 裁剪和深度缓存等内容；第 12～14 章讨论了高级 3D 渲染技术，包括透视修正纹理映射、Alpha 混合、1/z 缓存、纹理滤波、空间划分和可见性算法、阴影、光照映射等；第 15～16 章讨论了动画、运动碰撞检测和优化技术。

本书适合于有一定编程经验并想从事游戏编程工作或对 3D 图形学感兴趣的人员阅读。

作者简介

André LaMothe 有 25 年的计算行业从业经验，拥有数学、计算机科学和电子工程等学位，是 20 岁时就在 NASA 做研究工作的少数几人之一。在 30 岁之前，他在硅谷的众多公司中从事过咨询工作，了解了公司运作，获得了多种领域的知识，如电信、虚拟现实、机器人技术、编译器设计、3D 引擎、人工智能以及计算和工程的其他领域的知识。

他创办的公司 Xtreme Game 公司一直是自成一体的游戏开发、发行商。后来他创办了 Xtreme Games Developer Conference（XGDC），为游戏开发人员提供了费用更低廉的 GDC 替代品。

最近，他参与了多个项目的开发工作，其中包括 eGamezone Networks——一个公平、有趣、没有任何广告的网络游戏分发系统。他还创建了一家新公司——Nurve Networks 公司，为在乎价格的消费者和业余爱好者开发手持设备上的视频游戏系统。最后，他还是世界上最庞大的游戏开发系列丛书的编辑。

技术审校人员简介

David Franson 从 1990 年起开始从事网络、编程、2D/3D 计算机图形学等工作。2000 年，他辞去了纽约最大的娱乐律师事务所之一的信息系统总监职务，全身心地投入游戏开发工作。他还是 *2D Artwork and 3D Modeling for Game Artists* 的作者，当前正在编写 *Xbox Hackz and Modz* 一书。

献　　词

谨以此书献给所有诚实正直的人们——人间正道是沧桑。您的举动不会默默无闻……

致　谢

首先要感谢 Sams 出版社的全体员工。这里有必要说明一下，Sams 出版社有一套非常严格的图书出版流程，“大师技巧”系列图书都是完全按这种流程成书的。

感谢出版人 Michael Stephens、组稿编辑 Kim Spilker、可信赖的开发编辑 Mark Renfrow，当然还有确保一切工作顺利进行的项目编辑 George Nedeff。书中还有很多具体细节，这首先要感谢文字编辑 Seth Kerney，当然还有技术编辑 David Franson，他提供了包括摩托艇在内的一些 3D 模型以及演示程序中使用的众多纹理。

最后，本书如果没有附带光盘将是不完整的，感谢媒体开发人员 Dan Scherf；还要感谢无名英雄 Erika Millen 为本书制作了详细而完备的索引——这对于读者查找内容至关重要。

有一个特殊人物这里不能不提，那就是 Angela Kozlowski；她与我一道完成了本书前面的 1/4，帮助确定了整体框架，让所有其他人明白本书有多么重要和与众不同。在本书完成之前，她被调往其他岗位工作。

接下来要感谢那些为我提供、租借或帮助我获得软件和硬件的公司和个人。Intel 公司的 David Hackson 提供了最新的高级 C++和 Fortran 编译器以及 VTune， Caligari 公司的 Kristine Gardner 提供了各种版本的 trueSpace，微软公司的老朋友 Stacey Tsurusaki 给我提供了微软公司内部使用的尖端工具。当然，还有一些公司给我提供了其产品的评估版或允许我将其软件放到本书的附带光盘中，如 JASC 公司允许我使用 Paint Shop Pro，Sonic Foundry 允许我使用 Sound Forge，Right Hemisphere's Mary Alice Krayecki 提供了 Deep Exploration and Deep Paint UV 的拷贝。

接下来要感谢那些在我撰写本书期间提供帮助的朋友。我这个人“爱憎分明”，但正是这一点让我在聚会中很有趣！

回到正题，感谢 Mike Perone 帮助我获得“软件”并处理网络方面的问题；感谢 Mark Bell 耐心地倾听我抱怨——只有创业者才能理解我们经历的困境；感谢参与举办 Vintage 计算机节的 Sellam Ismail 提供大量非常棒且价格低廉的东西；感谢 John Romero 经常同我交流，让我相信这个行业中有那么有趣的人！

感谢 Nolan Bushnell 邀请我前往 uWink 公司并抽空与我讨论视频游戏，与创建 Atari 的人共度时光可是难得的机会！感谢他为本书作序。

另外，感谢我的新助手 Alex Varanese 容忍我喋喋不休地谈论尽善尽美、最后期限、超越极限、展望成功。

最后，感谢那些真正能够容忍我好斗、苛刻个性的人：母亲、父亲、耐心的女友 Anita，还有小狗 Ariel。

序

很荣幸应邀为这本向程序员介绍从零开始创建下一代视频游戏所需技能的重要著作做序。从绘制像素开始介绍创建实时 3D 引擎的著作并不多。以前，先进技术和 Atari 公司开发的粗糙游戏形成了鲜明的反差，对此进行反思时发现，我们确实曾致力于提高技术发展水平，但看起来收效甚微。

回顾过去，早期的游戏从技术的角度看根本算不上计算机游戏，它们不过是奇特的信号生成器，是使用计数器和基于布尔逻辑的移位寄存器的状态机，由拼凑而成的 MSI（中等规模集成）门组成。读者可能还记得我于上世纪 70 年代开发的第一款游戏——*Computer Space*，它面世的时间比 Intel 4004 早了 4 年，比 8080 早 6 年。我曾希望有微处理器来运行它！在那时候，对于任何重要的实时计算而言，典型的时钟速度太慢了。我们开发的第一款使用微处理器的游戏是 *Asteriods*，即使是这样，仍使用了大量的硬件来支持该程序，因为微处理器不能完成软件的所有工作。

当前，我们正在通向创建照片级图像的道路上迈进，虽然这种目标还未达到。能够动态地创建这样的图像确实令人兴奋。软件和硬件工具为游戏制作人员提供了非常强大的创建真实感游戏世界、环境和角色的功能。使用这些功能可以缩短游戏制作周期，增加财富，使开发新的游戏项目成为可能。

André LaMothe 不但谙熟这些技术，还有独特的“游戏感”。多年来，我见过很多精通尖端技术的专家，但缺乏编写优秀游戏必备的游戏感；而其他人有良好的游戏感，却是平庸的程序员。André不但是真正的游戏大师，还是软件大师，这一点在他撰写的每本著作中都表现得淋漓尽致。

在我们最近合作开发的一个项目中，André对尖端技术的精通、历史知识的广博以及对早期一些默默无闻游戏的了解之深，给我留下了深刻的印象。更值得一提的是，他竟然只花了 19 天的时间就为我编写了一款完整的游戏！知道一些成功案例很容易，但对败笔也了如指掌很难。是的，Atari 确实有一些糟糕的败笔，但如大家所知，我们也开发了很多著名的经典游戏。

希望读者喜欢本书，并以此为跳板，在未来开发出让我流连忘返的优秀游戏。

Atari 公司创始人

Nolan Bushnell

前　言

游戏编程原理和实践

很久以前，我编写了一本有关游戏编程的图书《Windows 游戏编程大师技巧》，终于实现了夙愿——为读者编写一本介绍如何制作游戏的图书。多年后，我在游戏编程方面的经验更加丰富，思想也更睿智，同时学会了更多游戏编程的技巧。读者即使没有阅读过《Windows 游戏编程大师技巧》，也能读懂本书；但需要提醒您的是，本书的内容更深，重点为 3D 游戏编程，要求读者具备众多的背景知识。

本书将续写和弥补《Windows 游戏编程大师技巧》没有涉及的内容，进一步探讨 3D 游戏编程的概念，在时间和篇幅允许的情况下，尽可能地涵盖 3D 游戏编程的每个重要主题。

当然，我不会假设读者是位大师级程序员且已经知道如何制作游戏。本书是为游戏编程新手编写的，针对的读者群是中高级程序员。如果读者不熟悉 C/C++编程，势必会在阅读本书的过程中深感迷惘。市面上有很多优秀的 C/C++图书，建议读者参考 Stephen Prata 或 Robert Lafore 的著作。在笔者看来，他们是世界上最棒的 C/C++图书作者。

当前是有史以来游戏行业最美好的时代。现在的技术足以让您创建出栩栩如生的游戏！想象一下即将出现的技术，PlayStation 2、Xbox 和 GameCube 都很酷。然而，这些技术掌握起来并不容易，您必须付出艰苦的努力。

当前，游戏编程的难度更高了，为制作游戏必须掌握更多的技能。然而，如果您正阅读这段文字，表明您乐于迎接挑战。算您找对了地方，阅读本书后，您将能够使用自己编写的软件光栅化模块，制作出支持纹理映射和光照效果的 3D PC 视频游戏。另外，您还将理解 3D 图形学的基本原理，更深入地认识与运用当前和未来的 3D 硬件。

内容简介

本书的内容极其丰富，涵盖了创建基于 Windows 9x/2000 的 PC 游戏所需的全部知识，其中包括以下主题：

- 《Windows 游戏编程大师技巧》中开发的引擎；
- Win32 编程和 DirectX 基础知识；
- 包括四元数在内的高等数学知识；
- 2D/3D 图形学和算法；
- 3D 投影和相机操控；
- 线框和实心模式渲染；
- 光照和纹理映射；
- 高级可见性算法；
- 3D 动画技术。

读者可能会问，本书介绍如何使用 3D 硬件还是使用代码实现 3D 软件光栅化模块？

答案是后者。只有懦弱的人才依赖于 3D 硬件，真正的游戏程序员能够从头开始编写 3D 引擎，对这样的工作充满激情，同时知道如何使用 3D 硬件。本书将介绍真正的 3D 游戏编程，具备这些知识后，读者将能够在两三周内学会使用任何 3D API。

在作者看来，如果您知道如何编写纹理映射函数和观察系统，使用硬件时将更为得心应手。另外，您不能假设每台计算机都配置了优秀的 3D 硬件，这样的时代还没有到来；只因为计算机没有 3D 硬件就将其排除在目标市场之外是非常糟糕的，尤其是在您没有数百万的资金，又想进入游戏市场时。在这种情况下，您将从非 3D 加速的软件市场着手。

最后，读者肯定对“Windows-DirectX”有些担心。只要方法得当，Windows 编程实际上非常简单、有趣，DOS32 编程中的很多问题都不再会出现。不要将 Windows 编程视为障碍——它让我们能够将更多的时间花在游戏代码而不是诸如 GUI、I/O 和图形驱动程序等细节上。如果想为市面上所有的 2D/3D 加速硬件编写图形驱动器，就是日夜不停地干也干不完，况且还有声卡、游戏杆等硬件呢。

读者必须具备的知识

本书假定读者有很强的编程技能。如果您不懂如何编写 C 语言代码，不知道怎样使用编译器，将会在阅读本书时感到相当迷惘。本书还使用了一些 C++代码，这可能会让 C 语言程序员感到有些担心。不过也不用怕，我在做任何怪异的事情前将提醒读者。如果您需要 C++程序设计的速成课程，可参阅附录 D。基本上，本书只有有关 DirectX 的范例偶尔使用了 C++。

然而，我还是决定在本书中稍微多用些 C++，因为在游戏编程中，很多东西都是面向对象的，如果将它们设计成 C 语言风格的结构，简直是悖理逆天。总之，如果您能够使用 C 语言进行编程，那很好；如果能够使用 C/C++进行编程，阅读本书将完全不成问题。

众所周知，计算机程序是由逻辑和数学计算组成的。3D 视频游戏的重点在数学运算，数学在 3D 图形学中几乎无处不在。幸运的是，读者只需具备一些基本的代数和几何学知识即可；本书还将介绍有关向量和矩阵的知识。读者只要知道加、减、乘、除，就可以理解 90%以上的内容，虽然不能亲自推导。毕竟，最终的目的只是要能使用其中的代码。

本书的组织结构

本书由 6 部分组成。

- 第一部分——3D 游戏编程简介。简要地介绍游戏、Windows 和 DirectX 编程，将建立一个虚拟计算机接口，用于创建所有的演示程序。
- 第二部分——3D 数学和变换。介绍各种数学概念，创建一个供本书使用的完整数学库。这部分的最后几章涉及 3D 图形学、数据结构、相机和线框模式渲染等。
- 第三部分——基本 3D 渲染。讨论光照、基本着色、隐藏面消除和 3D 裁剪。
- 第四部分——高级 3D 渲染。讨论纹理映射、高级光照、阴影以及 BSP 树、入口等空间划分算法等。
- 第五部分——高级动画、物理建模和优化。介绍动画、运动、碰撞检测、简单物理建模等。另外，还将讨论层次型建模、加载大型游戏世界和众多的优化技术。

附带光盘的内容

附带光盘包含本书全部的源代码、可执行文件、范例程序、素材、软件程序、音效和技术文章，其目录结构如下：

```
Tricks 3D\

SOURCE\
      T3DIICHAP01\
```

```
        T3DIICHAP02\
                .
                .
        T3DIICHAP16\

TOOLS\
GAMES\
MEDIA\
        BITMAPS\
        3DMODELS\
        SOUND\


DIRECTX\


ARTICLES\
```

每个主目录都包含您所需的特定数据，具体情况如下。

- Tricks 3D：包含其他所有目录的根目录。请阅读 README.TXT 文件以便了解最后的修改。
- SOURCE：按章节顺序收录了书中所有的源代码。只需将整个 SOURCE\目录拷贝到硬盘上就可以使用。
- TOOLS：收录了各公司慷慨地允许我放入本光盘的演示版程序。
- MEDLA：可在您的游戏中随便使用的图像、声音和模型。
- DIRECTX：最新版本的 DirectX SDK。
- GAMES：大量演示了软件光栅化的共享版 2D、3D 游戏。
- ARTICLES：由 3D 游戏编程领域的许多老手撰写的启迪性文章。

附带光盘包含各种程序和数据，因此没有统一的安装程序，您需要自行安装不同的程序和数据。然而，在大多数情况下，只需将 SOURCE\目录拷贝到硬盘中就可以了。至于其他程序和数据，可以在需要时安装它们。

安装 DirectX

附带光盘中最重要的、必须安装的部分是 DirectX SDK 及其运行阶段文件。安装程序位于 DIRECTX\目录中，该目录中还有一个 README.TXT 文件，阐明了最后的修改。

注意：必须安装了 DirectX 8.1 SDK 或更高版本（附带光盘中提供了 DirectX 9.0）才能使用本书的源代码。如果不能确定系统中是否已经安装了最新版本的 DirectX SDK，请运行安装程序进行确认。

编译程序

本书的程序是使用 Microsoft Visual C++ 6.0 编写的。然而，多数情况下，也可以任何与 Win32 兼容的编译器进行编译。尽管如此，我还是推荐使用 Microsoft VC++或.NET，因为用它们做这类工作最有效率。

如果您不熟悉您的编译器集成开发环境（IDE），编译 Windows 程序时肯定会遇到麻烦。因此，编译程序之前，请务必花些时间来熟悉编译器，至少达到知道如何编译控制台（console）程序“Hello World”的程度。

要编译生成 Windows Win32.EXE 程序，只需将工程的目标程序设置为 Win32.EXE，再进行编译。然而，要创建 DirectX 程序，必须在工程中包含 DirectX 导入库。您可能认为只要将 DirectX 库添加到包含路径（Include path）中即可，但这样不行。为避免麻烦，最好手工将 DirectX .LIB 文件包含到工程中，.LIB 文件位于 DirectX SDK 安装目录中的 LIB\目录下。这样将不会出现链接错误。在大多数情况下，需要下面这些文件。

- DDRAW.LIB：DirectDraw 导入库。

- DINPUT.LIB：DirectInput 导入库。
- DINPUT8.LIB：DirectInput8 导入库。
- DSOUND.LIB：DirectSound 导入库。
- WINMM.LIB： Windows 多媒体扩展库。

具体使用上述文件时，将更详细地介绍它们；当链接器指出“未知符号（Unresolved Symbol）”错误时，请检查是否包含了这些库。我不想从新手那里再收到有关这方面的电子邮件。

除 DirectX .LIB 文件外，还需要将 DirectX .H 文件放到头文件搜索路径中。另外，请务必将 DirectX SDK 目录放在搜索路径列表的最前面，因为很多 C++编译器带有旧版本的 DirectX，编译器可能在其 INCLUDE\ 目录下找到旧版本的头文件，而使用这些头文件是错误的。正确的位置是 DirectX SDK 的包含目录，即 DirectX SDK 安装目录中的 INCLUDE\目录。

最后，如果读者使用的是 Borland 产品，请务必使用 Borland 版本的 DirectX .LIB 文件，它们位于 DirectX SDK 安装目录中的 BORLAND\目录下。

目　录

第一部分　3D 游戏编程简介

第 1 章　3D 游戏编程入门 ……2

1.1　简介 ……2
1.2　2D/3D 游戏的元素 ……3
1.2.1　初始化 ……3
1.2.2　进入游戏循环 ……3
1.2.3　读取玩家输入 ……4
1.2.4　执行 AI 和游戏逻辑 ……4
1.2.5　渲染下一帧 ……4
1.2.6　同步显示 ……4
1.2.7　循环 ……4
1.2.8　关闭 ……5
1.3　通用游戏编程指南 ……7
1.4　使用工具 ……9
1.4.1　3D 关卡编辑器 ……12
1.4.2　使用编译器 ……13
1.5　一个 3D 游戏范例：*Raiders 3D* ……15
1.5.1　事件循环 ……33
1.5.2　核心 3D 游戏逻辑 ……34
1.5.3　3D 投影 ……35
1.5.4　星空 ……36
1.5.5　激光炮和碰撞检测 ……37
1.5.6　爆炸 ……37
1.5.7　玩 *Raiders3D* ……37
1.6　总结 ……37

第 2 章　Windows 和 DirectX 简明教程 ……38

2.1　Win32 编程模型 ……38
2.2　Windows 程序的最小需求 ……39
2.3　一个基本的 Windows 应用程序 ……43
2.3.1　Windows 类 ……43
2.3.2　注册 Windows 类 ……47
2.3.3　创建窗口 ……47
2.3.4　事件处理程序 ……48
2.3.5　主事件循环 ……52
2.3.6　构建实时事件循环 ……55
2.4　DirectX 和 COM 简明教程 ……56
2.4.1　HEL 和 HAL ……57
2.4.2　DirectX 基本类 ……58
2.5　COM 简介 ……59
2.5.1　什么是 COM 对象 ……60

2.5.2 创建和使用 DirectX COM 接口……61
2.5.3 查询接口……62
2.6 总结……64

第 3 章 使用虚拟计算机进行 3D 游戏编程……65

3.1 虚拟计算机接口简介……65
3.2 建立虚拟计算机接口……66
3.2.1 帧缓存和视频系统……66
3.2.2 使用颜色……70
3.2.3 缓存交换……71
3.2.4 完整的虚拟图形系统……73
3.2.5 I/O、声音和音乐……73
3.3 T3DLIB 游戏控制台……74
3.3.1 T3DLIB 系统概述……74
3.3.2 基本游戏控制台……74
3.4 T3DLIB1 库……79
3.4.1 DirectX 图形引擎体系结构……79
3.4.2 基本常量……79
3.4.3 工作宏……81
3.4.4 数据类型和结构……81
3.4.5 函数原型……84
3.4.6 全局变量……88
3.4.7 DirectDraw 接口……89
3.4.8 2D 多边形函数……92
3.4.9 数学函数和错误函数……97
3.4.10 位图函数……99
3.4.11 8 位调色板函数……102
3.4.12 实用函数……104
3.4.13 BOB（Blitter 对象）引擎……106
3.5 T3DLIB2 DirectX 输入系统……112
3.6 T3DLIB3 声音和音乐库……116
3.6.1 头文件……117
3.6.2 类型……117
3.6.3 全局变量……117
3.6.4 DirectSound API 封装函数……118
3.6.5 DirectMusic API 封装函数……121
3.7 建立最终的 T3D 游戏控制台……124
3.7.1 映射真实图形到虚拟接口的非真实图形……124
3.7.2 最终的 T3DLIB 游戏控制台……126
3.8 范例 T3LIB 应用程序……134
3.8.1 窗口应用程序……134
3.8.2 全屏应用程序……135
3.8.3 声音和音乐……136
3.8.4 处理输入……136
3.9 总结……139

第二部分 3D 数学和变换

第 4 章 三角学、向量、矩阵和四元数……142

4.1 数学表示法……142
4.2 2D 坐标系……143
4.2.1 2D 笛卡尔坐标……143
4.2.2 2D 极坐标……144
4.3 3D 坐标系……147
4.3.1 3D 笛卡尔坐标……147
4.3.2 3D 柱面坐标……149
4.3.3 3D 球面坐标……150
4.4 三角学……151
4.4.1 直角三角形……151
4.4.2 反三角函数……153
4.4.3 三角恒等式……153
4.5 向量……154
4.5.1 向量长度……155
4.5.2 归一化……155
4.5.3 向量和标量的乘法……155
4.5.4 向量加法……156
4.5.5 向量减法……157
4.5.6 点积……157
4.5.7 叉积……159
4.5.8 零向量……160
4.5.9 位置和位移向量……160
4.5.10 用线性组合表示的向量……161
4.6 矩阵和线性代数……161
4.6.1 单位矩阵……162
4.6.2 矩阵加法……163
4.6.3 矩阵的转置……163
4.6.4 矩阵乘法……164
4.6.5 矩阵运算满足的定律……165
4.7 逆矩阵和方程组求解……165
4.7.1 克来姆法则……167
4.7.2 使用矩阵进行变换……168
4.7.3 齐次坐标……169
4.7.4 应用矩阵变换……170
4.8 基本几何实体……176
4.8.1 点……176
4.8.2 直线……176
4.8.3 平面……179
4.9 使用参数化方程……182
4.9.1 2D 参数化直线……182
4.9.2 3D 参数化直线……184
4.10 四元数简介……189
4.10.1 复数理论……189
4.10.2 超复数……193
4.10.3 四元数的应用……197
4.11 总结……200

第 5 章 建立数学引擎……201

5.1 数学引擎概述……201
5.1.1 数学引擎的文件结构……201
5.1.2 命名规则……202
5.1.3 错误处理……203
5.1.4 关于 C++的最后说明……203

5.2 数据结构和类型……203
5.2.1 向量和点……203
5.2.2 参数化直线……204
5.2.3 3D 平面……206
5.2.4 矩阵……206
5.2.5 四元数……209
5.2.6 角坐标系支持……210
5.2.7 2D 极坐标……210
5.2.8 3D 柱面坐标……211
5.2.9 3D 球面坐标……211
5.2.10 定点数……212
5.3 数学常量……213
5.4 宏和内联函数……214
5.4.1 通用宏……218
5.4.2 点和向量宏……218
5.4.3 矩阵宏……219
5.4.4 四元数……220
5.4.5 定点数宏……221
5.5 函数原型……221
5.6 全局变量……224
5.7 数学引擎 API 清单……225
5.7.1 三角函数……225
5.7.2 坐标系支持函数……226
5.7.3 向量支持函数……228
5.7.4 矩阵支持函数……235
5.7.5 2D 和 3D 参数化直线支持函数……245
5.7.6 3D 平面支持函数……248
5.7.7 四元数支持函数……252
5.7.8 定点数支持函数……259
5.7.9 方程求解支持函数……263
5.8 浮点单元运算初步……265
5.8.1 FPU 体系结构……266
5.8.2 FPU 堆栈……266
5.8.3 FPU 指令集……268
5.8.4 经典指令格式……270
5.8.5 内存指令格式……271
5.8.6 寄存器指令格式……271
5.8.7 寄存器弹出指令格式……271
5.8.8 FPU 范例……271
5.8.9 FLD 范例……272
5.8.10 FST 范例……272
5.8.11 FADD 范例……273
5.8.12 FSUB 范例……275
5.8.13 FMUL 范例……276
5.8.14 FDIV 范例……278
5.9 数学引擎使用说明……279
5.10 关于数学优化的说明……280
5.11 总结……280

第 6 章 3D 图形学简介……282

6.1 3D 引擎原理……282
6.2 3D 游戏引擎的结构……282
6.2.1 3D 引擎……283
6.2.2 游戏引擎……283
6.2.3 输入系统和网络……284
6.2.4 动画系统……284
6.2.5 碰撞检测和导航系统……287
6.2.6 物理引擎……288
6.2.7 人工智能系统……289
6.2.8 3D 模型和图像数据库……289
6.3 3D 坐标系……291
6.3.1 模型（局部）坐标……291
6.3.2 世界坐标……293
6.3.3 相机坐标……296
6.3.4 有关相机坐标的说明……302
6.3.5 隐藏物体（面）消除和裁剪……303
6.3.6 透视坐标……308
6.3.7 流水线终点：屏幕坐标……315
6.4 基本的 3D 数据结构……321
6.4.1 表示 3D 多边形数据时需要考虑的问题……322
6.4.2 定义多边形……323
6.4.3 定义物体……327
6.4.4 表示世界……330
6.5 3D 工具……331
6.6 从外部加载数据……332
6.6.1 PLG 文件……333
6.6.2 NFF 文件……335
6.6.3 3D Studio 文件……338
6.6.4 Caligari COB 文件……343
6.6.5 Microsoft DirectX .X 文件……345
6.6.6 3D 文件格式小结……345
6.7 基本刚性变换和动画……345
6.7.1 3D 平移……345
6.7.2 3D 旋转……346
6.7.3 3D 变形……347
6.8 再看观察流水线……348
6.9 3D 引擎类型……349
6.9.1 太空引擎……349
6.9.2 地形引擎……350
6.9.3 FPS 室内引擎……351
6.9.4 光线投射和体素引擎……352
6.9.5 混合引擎……353
6.10 将各种功能集成到引擎中……353
6.11 总结……353

第 7 章 渲染 3D 线框世界……354

7.1 线框引擎的总体体系结构……354
7.1.1 数据结构和 3D 流水线……355
7.1.2 主多边形列表……357
7.1.3 新的软件模块……359
7.2 编写 3D 文件加载器……359
7.3 构建 3D 流水线……367

7.3.1 通用变换函数……367
7.3.2 局部坐标到世界坐标变换……372
7.3.3 欧拉相机模型……375
7.3.4 UVN 相机模型……377
7.3.5 世界坐标到相机坐标变换……387
7.3.6 物体剔除……390
7.3.7 背面消除……393
7.3.8 相机坐标到透视坐标变换……395
7.3.9 透视坐标到屏幕（视口）坐标变换……399
7.3.10 合并透视变换和屏幕变换……403
7.4 渲染 3D 世界……405
7.5 3D 演示程序……408
7.5.1 单个 3D 三角形……408
7.5.2 3D 线框立方体……411
7.5.3 消除了背面的 3D 线框立方体……413
7.5.4 3D 坦克演示程序……414
7.5.5 相机移动的 3D 坦克演示程序……416
7.5.6 战区漫步演示程序……418
7.6 总结……421

第三部分 基本 3D 渲染

第 8 章 基本光照和实体造型……424

8.1 计算机图形学的基本光照模型……424
8.1.1 颜色模型和材质……426
8.1.2 光源类型……432
8.2 三角形的光照计算和光栅化……437
8.2.1 为光照做准备……441
8.2.2 定义材质……442
8.2.3 定义光源……445
8.3 真实世界中的着色……449
8.3.1 16 位着色……449
8.3.2 8 位着色……450
8.3.3 一个健壮的用于 8 位模式的 RGB 模型……450
8.3.4 一个简化的用于 8 位模式的强度模型……453
8.3.5 固定着色……457
8.3.6 恒定着色……459
8.3.7 Gouraud 着色概述……472
8.3.8 Phong 着色概述……474
8.4 深度排序和画家算法……475
8.5 使用新的模型格式……479
8.5.1 分析器类……479
8.5.2 辅助函数……482
8.5.3 3D Studio MAX ASCII 格式.ASC……484
8.5.4 TrueSpace ASCII.COB 格式……486
8.5.5 Quake II 二进制.MD2 格式概述……494
8.6 3D 建模工具简介……495
8.7 总结……497

第 9 章 插值着色技术和仿射纹理映射……498

9.1 新 T3D 引擎的特性……498
9.2 更新 T3D 数据结构和设计……499
9.2.1 新的#defines……499
9.2.2 新增的数学结构……501
9.2.3 实用宏……502
9.2.4 添加表示 3D 网格数据的特性……503
9.2.5 更新物体结构和渲染列表结构……508
9.2.6 函数清单和原型……511
9.3 重新编写物体加载函数……517
9.3.1 更新.PLG/PLX 加载函数……517
9.3.2 更新 3D Studio .ASC 加载函数……527
9.3.3 更新 Caligari .COB 加载函数……528
9.4 回顾多边形的光栅化……532
9.4.1 三角形的光栅化……532
9.4.2 填充规则……535
9.4.3 裁剪……537
9.4.4 新的三角形渲染函数……538
9.4.5 优化……542
9.5 实现 Gouraud 着色处理……543
9.5.1 没有光照时的 Gouraud 着色……544
9.5.2 对使用 Gouraud Shader 的多边形执行光照计算……553
9.6 基本采样理论……560
9.6.1 一维空间中的采样……560
9.6.2 双线性插值……561
9.6.3 u 和 v 的插值……563
9.6.4 实现仿射纹理映射……564
9.7 更新光照/光栅化引擎以支持纹理……566
9.8 对 8 位和 16 位模式下优化策略的最后思考……571
9.8.1 查找表……571
9.8.2 网格的顶点结合性……572
9.8.3 存储计算结果……572
9.8.4 SIMD……573
9.9 最后的演示程序……573
9.10 总结……576

第 10 章 3D 裁剪……577

10.1 裁剪简介……577
10.1.1 物体空间裁剪……577
10.1.2 图像空间裁剪……580
10.2 裁剪算法……581
10.2.1 有关裁剪的基本知识……581
10.2.2 Cohen-Sutherland 裁剪算法……585

10.2.3 Cyrus-Beck/梁友栋-Barsky 裁剪算法……586
10.2.4 Weiler-Atherton 裁剪算法……588
10.2.5 深入学习裁剪算法……590
10.3 实现视景体裁剪……591
10.3.1 几何流水线和数据结构……592
10.3.2 在引擎中加入裁剪功能……593
10.4 地形小议……611
10.4.1 地形生成函数……612
10.4.2 生成地形数据……619
10.4.3 沙地汽车演示程序……619
10.5 总结……623

第 11 章 深度缓存和可见性……624

11.1 深度缓存和可见性简介……624
11.2 z 缓存基础……626
11.2.1 z 缓存存在的问题……627
11.2.2 z 缓存范例……627
11.2.3 平面方程法……630
11.2.4 z 坐标插值……631
11.2.5 z 缓存中的问题和 1/z 缓存……632
11.2.6 一个通过插值计算 z 和 1/z 的例子……633
11.3 创建 z 缓存系统……635
11.4 可能的 z 缓存优化……649
11.4.1 使用更少的内存……649
11.4.2 降低清空 z 缓存的频率……650
11.4.3 混合 z 缓存……651
11.5 z 缓存存在的问题……651
11.6 软件和 z 缓存演示程序……652
11.6.1 演示程序 I：z 缓存可视化……652
11.6.2 演示程序 II：Wave Raider……653
11.7 总结……658

第四部分　高级 3D 渲染

第 12 章 高级纹理映射技术……660

12.1 纹理映射——第二波……660
12.2 新的光栅化函数……667
12.2.1 最终决定使用定点数……667
12.2.2 不使用 z 缓存的新光栅化函数……668
12.2.3 支持 z 缓存的新光栅化函数……670
12.3 使用 Gouruad 着色的纹理映射……671
12.4 透明度和 alpha 混合……677
12.4.1 使用查找表来进行 alpha 混合……678
12.4.2 在物体级支持 alpha 混合功能……688
12.4.3 在地形生成函数中加入 alpha 支持……694
12.5 透视修正纹理映射和 1/z 缓存……696
12.5.1 透视纹理映射的数学基础……696
12.5.2 在光栅化函数中加入 1/z 缓存功能……702
12.5.3 实现完美透视修正纹理映射……707
12.5.4 实现线性分段透视修正纹理映射……710
12.5.5 透视修正纹理映射的二次近似……714
12.5.6 使用混合方法优化纹理映射……718
12.6 双线性纹理滤波……719
12.7 Mipmapping 和三线性纹理滤波……724
12.7.1 傅立叶分析和走样简介……725
12.7.2 创建 Mip 纹理链……727
12.7.3 选择 mip 纹理……734
12.7.4 三线性滤波……739
12.8 多次渲染和纹理映射……740
12.9 使用单个函数来完成渲染工作……741
12.9.1 新的渲染场境……741
12.9.2 设置渲染场境……743
12.9.3 调用对渲染场境进行渲染的函数……745
12.10 总结……753

第 13 章 空间划分和可见性算法……754

13.1 新的游戏引擎模块……754
13.2 空间划分和可见面判定简介……754
13.3 二元空间划分……757
13.3.1 平行于坐标轴的二元空间划分……758
13.3.2 任意平面空间划分……759
13.3.3 使用多边形所在的平面来划分空间……760
13.3.4 显示/访问 BSP 树中的每个节点……762
13.3.5 BSP 树数据结构和支持函数……763
13.3.6 创建 BSP 树……765
13.3.7 分割策略……767
13.3.8 遍历和显示 BSP 树……775
13.3.9 将 BSP 树集成到图形流水线中……784
13.3.10 BSP 关卡编辑器……785
13.3.11 BSP 的局限性……793
13.3.12 使用 BSP 树的零重绘策略……794
13.3.13 将 BSP 树用于剔除……795
13.3.14 将 BSP 树用于碰撞检测……802
13.3.15 集成 BSP 树和标准渲染……802
13.4 潜在可见集……807
13.4.1 使用潜在可见集……808
13.4.2 潜在可见集的其他编码方法……809
13.4.3 流行的 PVS 计算方法……810

13.5 入口……811
13.6 包围体层次结构和八叉树……813
13.6.1 使用 BHV 树……815
13.6.2 运行性能……816
13.6.3 选择策略……817
13.6.4 实现 BHV……818
13.6.5 八叉树……825
13.7 遮掩剔除……825
13.7.1 遮掩体……826
13.7.2 选择遮掩物……826
13.7.3 混合型遮掩物选择方法……827
13.8 总结……827

第 14 章 阴影和光照映射……828

14.1 新的游戏引擎模块……828
14.2 概述……828
14.3 简化的阴影物理学……829
14.4 使用透视图像和广告牌来模拟阴影……832
14.4.1 编写支持透明功能的光栅化函数……833
14.4.2 新的库模块……835
14.4.3 简单阴影……837
14.4.4 缩放阴影……839
14.4.5 跟踪光源……841
14.4.6 有关模拟阴影的最后思考……844
14.5 平面网格阴影映射……845
14.5.1 计算投影变换……845
14.5.2 优化平面阴影……848
14.6 光照映射和面缓存技术简介……848
14.6.1 面缓存技术……850
14.6.2 生成光照图……850
14.6.3 实现光照映射函数……851
14.6.4 暗映射（dark mapping）……853
14.6.5 光照图特效……854
14.6.6 优化光照映射代码……854
14.7 整理思路……854
14.8 总结……854

第五部分 高级动画、物理建模和优化

第 15 章 3D 角色动画、运动和碰撞检测……858

15.1 新的游戏引擎模块……858
15.2 3D 动画简介……858
15.3 Quake II .MD2 文件格式……859
15.3.1 .MD2 文件头……861
15.3.2 加载 Quake II .MD2 文件……868
15.3.3 使用.MD2 文件实现动画……874
15.3.4 .MD2 演示程序……882
15.4 不基于角色的简单动画……883
15.4.1 旋转运动和平移运动……883
15.4.2 复杂的参数化曲线移动……885
15.4.3 使用脚本来实现运动……885
15.5 3D 碰撞检测……887
15.5.1 包围球和包围圆柱……887
15.5.2 使用数据结构来提高碰撞检测的速度……888
15.5.3 地形跟踪技术……889
15.6 总结……890

第 16 章 优化技术……891

16.1 优化技术简介……891
16.2 使用 Microsoft Visual C++和 Intel VTune 剖析代码……892
16.2.1 使用 Visual C++进行剖析……892
16.2.2 分析剖析数据……893
16.2.3 使用 VTune 进行优化……894
16.3 使用 Intel C++编译器……899
16.3.1 下载 Intel 的优化编译器……900
16.3.2 使用 Intel 编译器……900
16.3.3 使用编译器选项……901
16.3.4 手工为源文件选择编译器……901
16.3.5 优化策略……902
16.4 SIMD 编程初步……902
16.4.1 SIMD 基本体系结构……903
16.4.2 使用 SIMD……903
16.4.3 一个 SIMD 3D 向量类……912
16.5 通用优化技巧……918
16.5.1 技巧 1：消除_ftol()……918
16.5.2 技巧 2：设置 FPU 控制字……918
16.5.3 技巧 3：快速将浮点变量设置为零……919
16.5.4 技巧 4：快速计算平方根……919
16.5.5 技巧 5：分段线性反正切……920
16.5.6 技巧 6：指针递增运算……920
16.5.7 技巧 7：尽可能将 if 语句放在循环外面……921
16.5.8 技巧 8：支化（branching）流水线……921
16.5.9 技巧 9：数据对齐……921
16.5.10 技巧 10：将所有简短函数都声明为内联的……922
16.5.11 参考文献……922
16.6 总结……922

第 1 章　3D 游戏编程入门

本章介绍一些基本的游戏编程主题，如游戏循环、2D 和 3D 游戏的差别等。最后将介绍一个很小的 3D 游戏，以帮助您正确地设置编译器和 DirectX。如果读者阅读过我的《Windows 游戏编程大师技巧》，可以考虑略读本章，并重点关注最后部分的内容；否则，即使您已经是一个中级或高级游戏程序员，也最好仔细阅读本章。本章主要包括下列内容：

- 简介；
- 2D/3D 游戏的元素；
- 游戏编程通用指南；
- 使用工具；
- 一个 16 位窗口游戏范例：*Raiders 3D*。

1.1　简　　介

本书是我写作的 2D 和 3D 游戏编程系列的第二卷。第一卷《Windows 游戏编程大师技巧》主要介绍了如下内容：

- Windows 编程；
- Win32 API；
- DirectX 基础；
- 人工智能；
- 基本物理模型；
- 声音和音乐；
- 算法；
- 游戏编程；
- 2D 光栅和矢量图形。

本书是前一卷的续写。但是，即使您没有阅读过前一卷，仍可以从本书获得许多与实时 3D 图形相关的知识，以更好地进行 3D 编程。因此，本书将主要适合那些阅读过第一卷且对 3D 游戏编程感兴趣，或者已经是 2D 程序员且想从软件和算法的角度学习 3D 技术的读者。

考虑到这一点，本书将重点介绍 3D 数学和图形学，而对于与游戏编程有关的内容只作简要介绍。我认为您已经了解了这些内容——如果您还不了解这些内容，建议先阅读《Windows 游戏编程大师技巧》或其他优秀的游戏编程书籍，并在计算机上进行练习，以熟悉 Windows、DirectX 和游戏编程。

另一方面，即使您没有阅读过《Windows 游戏编程大师技巧》，且不了解游戏编程，仍可以从本书学习到一些知识，因为这是一本有关图形学的书籍。我们将在每章中逐步建立 3D 引擎，但是为了节省时间，我们将以《Windows 游戏编程大师技巧》中开发的基本 DirectX 引擎为基础。它将有助于处理 2D、声音、音乐、输入等。当然，该引擎是一个 DirectX 7/8 引擎，而 DirectX 8.0 以上版本所做的修改使 2D 编程更困

第 1 章　3D 游戏编程入门

本章介绍一些基本的游戏编程主题，如游戏循环、2D 和 3D 游戏的差别等。最后将介绍一个很小的 3D 游戏，以帮助您正确地设置编译器和 DirectX。如果读者阅读过我的《Windows 游戏编程大师技巧》，可以考虑略读本章，并重点关注最后部分的内容；否则，即使您已经是一个中级或高级游戏程序员，也最好仔细阅读本章。本章主要包括下列内容：

- 简介；
- 2D/3D 游戏的元素；
- 游戏编程通用指南；
- 使用工具；
- 一个 16 位窗口游戏范例：*Raiders 3D*。

1.1　简　　介

本书是我写作的 2D 和 3D 游戏编程系列的第二卷。第一卷《Windows 游戏编程大师技巧》主要介绍了如下内容：

- Windows 编程；
- Win32 API；
- DirectX 基础；
- 人工智能；
- 基本物理模型；
- 声音和音乐；
- 算法；
- 游戏编程；
- 2D 光栅和矢量图形。

本书是前一卷的续写。但是，即使您没有阅读过前一卷，仍可以从本书获得许多与实时 3D 图形相关的知识，以更好地进行 3D 编程。因此，本书将主要适合那些阅读过第一卷且对 3D 游戏编程感兴趣，或者已经是 2D 程序员且想从软件和算法的角度学习 3D 技术的读者。

考虑到这一点，本书将重点介绍 3D 数学和图形学，而对于与游戏编程有关的内容只作简要介绍。我认为您已经了解了这些内容——如果您还不了解这些内容，建议先阅读《Windows 游戏编程大师技巧》或其他优秀的游戏编程书籍，并在计算机上进行练习，以熟悉 Windows、DirectX 和游戏编程。

另一方面，即使您没有阅读过《Windows 游戏编程大师技巧》，且不了解游戏编程，仍可以从本书学习到一些知识，因为这是一本有关图形学的书籍。我们将在每章中逐步建立 3D 引擎，但是为了节省时间，我们将以《Windows 游戏编程大师技巧》中开发的基本 DirectX 引擎为基础。它将有助于处理 2D、声音、音乐、输入等。当然，该引擎是一个 DirectX 7/8 引擎，而 DirectX 8.0 以上版本所做的修改使 2D 编程更困

难，因为 DirectDraw 已被集成到 Direct3D 中。我们将继续使用 DirectX 7/8 的接口，但在 DirectX 9.0 SDK 下进行编译。

注意：本书介绍软件和算法，而不是 DirectX，因此将在 DOS 下编写引擎。只是需要适当的高级图形、I/O 和声音层，因此 DirectX 8.0+是一个合理的选择。如果读者是 Linux 用户，该引擎移植到 SDL 也是相当容易的！

因而，如果您阅读过《Windows 游戏编程大师技巧》，将很熟悉该引擎及其函数；否则，可将其视为一个黑盒 API（当然其中包括源代码），因为本书将逐步在该引擎中加入 3D 功能。

不用担心，下一章将介绍整个 2D 引擎及其结构，您将知道每一个函数的功能；我还将提供一些演示程序，帮助您熟悉使用《Windows 游戏编程大师技巧》中创建的基本 DirectX 引擎。

本书将从一种通用的角度来介绍图形系统。本书的代码将以《Windows 游戏编程大师技巧》中创建的引擎为基础，该引擎是以标准的线性寻址方式建立一个双缓存图形系统。

要将代码移植到 Mac 或 Linux 平台上，只需要几小时或一个周末。本书的目标是以一种通用的角度介绍 3D 图形学和数学。DirectX 所处的 Windows 平台是当前最流行的计算机操作系统，因此将以它为基础进行介绍。本书的重点是高级的概念。

我编写过大约 20 个 3D 引擎，它们都保存在我的硬盘中。但是在编写本书时，我想随着本书的进度编写一个新的 3D 引擎，因此是为本书编写一个引擎，而不是编写一本关于引擎的书。本书不是讨论 3D 引擎，而是介绍如何编写 3D 引擎。因此，我并不清楚最后将得到一个什么样的 3D 引擎！实际上，我对于将会看到什么感到非常兴奋。我争取编写一个类似于 Quake 的 6DOF（degree of freedom，自由度）引擎，但谁知道结果将如何呢？您将学习创建 Quake 引擎所需要的所有知识，但我将重点介绍室外引擎或其他内容——谁知道呢。关键是本书将提供一个学习过程，而不是充满了作者注释的典型代码。我想您很清楚我在说什么！

最后，阅读过《Windows 游戏编程大师技巧》的读者可能还会注意到，本书中有些地方与该书相同。实际上，我没有办法一点也不重复前一本书中的内容，而直接介绍 3D 图形学。我不能假定每个人都购买了《Windows 游戏编程大师技巧》，也不能强迫他们去购买。总之，本书的前几章包含一些与 DirectX、引擎和 Windows 相关的内容。

1.2　2D/3D 游戏的元素

首先回顾一下视频游戏程序与其他程序的区别。视频游戏是一些相当复杂的软件。毫无疑问，它们是最难编写的程序。的确，编写类似于 MS Word 之类的软件要比编写星星游戏困难，但编写类似于 Unreal、Quake Arena 或 Halo 之类的游戏要比编写我所能想象到的任何其他程序都困难——包括军队武器控制软件！

这意味着您必须学习一种有利于实时应用程序和仿真的全新编程方式，而不是您习惯的单线、事件驱动或顺序逻辑程序。一个视频游戏本质上是一个持续不断的循环，它执行逻辑并在屏幕上绘制图像——通常以 30～60 帧/秒或更高的速度进行绘制。这类似于电影放映方式，不同的是您将控制电影的情节发展。

首先来看一个简化的游戏循环，如图 1.1 所示。

接下来的几个小节将介绍游戏循环的各个部分。

1.2.1　初始化

这部分执行与其他任何程序类似的标准操作，如内存分配、资源获取、从磁盘加载数据等。

1.2.2　进入游戏循环

这部分进入主游戏循环。用户将在这里不断地执行动作，直到退出主循环为止。

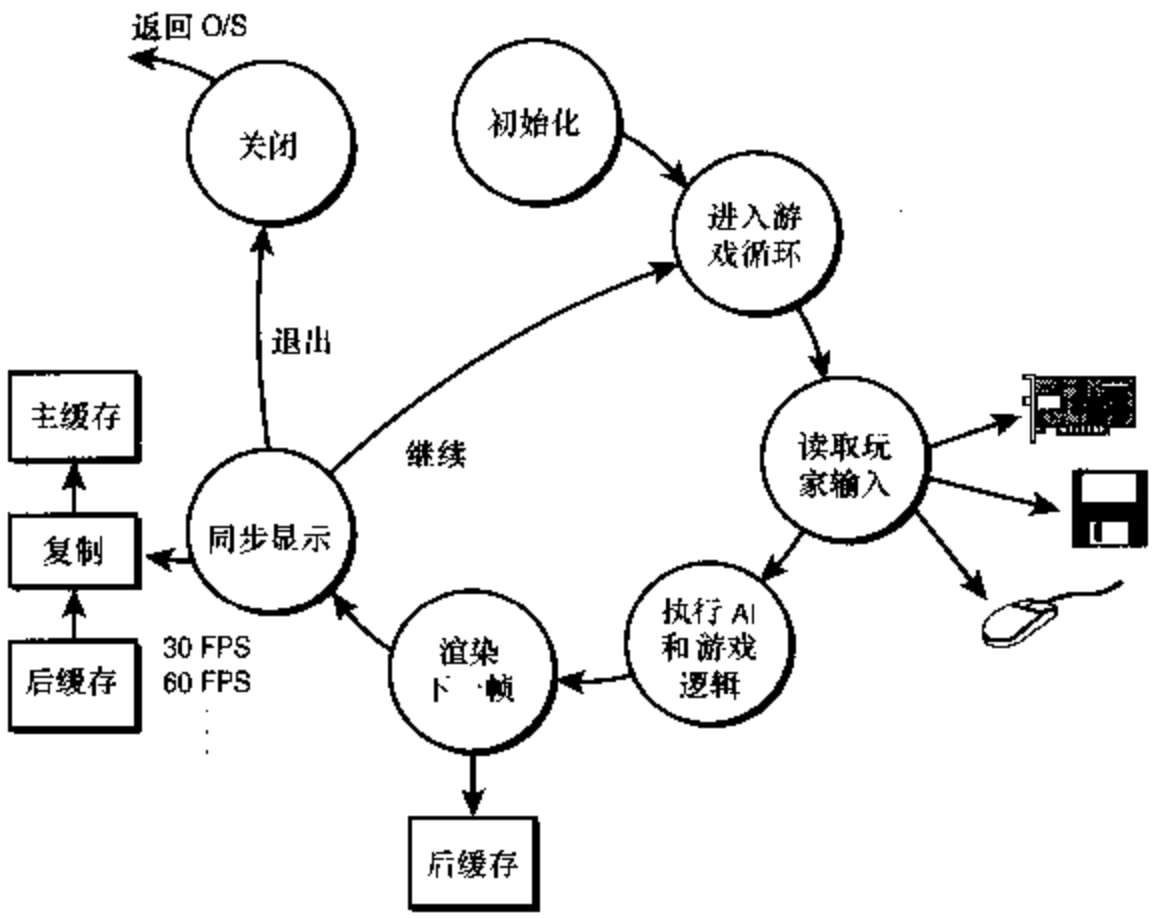


图 1.1　通用游戏循环体系结构

1.2.3　读取玩家输入

这部分处理玩家输入，或将其存储到缓冲区中，供 AI 和游戏逻辑使用。

1.2.4　执行 AI 和游戏逻辑

这部分包含游戏代码的主体部分，将执行人工智能、物理系统和通用游戏逻辑，并根据结果在屏幕上绘制下一帧。

1.2.5　渲染下一帧

在这一部分中，将根据玩家输入以及游戏 AI 和逻辑的执行结果，生成下一个游戏动画帧。这种图像通常在离屏（offscreen）缓存中绘制，因此用户无法看到渲染过程。下一帧图像被快速复制到可视区域，产生动画影像。在基于 3D 软件的引擎中，将由一个非常复杂的 3D 图形流水线来渲染构成世界的成千上万（甚至数百万）个多边形；在基于 OpenGL 或 Direct3D 的 3D 硬件加速引擎中，大部分工作都由硬件来承担。

1.2.6　同步显示

很多计算机的速度将因当前执行的游戏的复杂度而提高或降低。例如，如果屏幕上有 1000 个物体，CPU 的负载将比只有 10 个物体时重。因而，游戏的帧频将改变，这是不可接受的。因此，必须使用定时函数或等待函数，同步游戏到某个最大帧频，并保持该帧频。通常，30 帧/秒是可接受的最小帧频，60 帧/秒是比较理想的帧频。将帧频提高到 60 帧/秒以上没有多大意义，因为人的大脑很难处理超过 60 帧/秒的信息。

注意：尽管有些游戏在理论上可以达到 30～60 帧/秒的速率，但帧频会随着 3D 渲染的复杂度增加而下降。如果在代码的游戏逻辑、物理模型和 AI 部分使用基于时间的计算，将可在某种程度上保持显示速率尽可能地稳定。也就是说，如果在运动和动画控制中使用时间（t）进行计算，将得到显示速度更加稳定的动画。

1.2.7　循环

这部分相当简单，只需返回到游戏循环的开头，然后重新执行整个循环。

者看到的，状态切换是一个序列化过程。

1.3　通用游戏编程指南

下面介绍通用游戏编程技术和思想，您应该学习并尽可能采用这些技术和思想，从而更容易地进行游戏编程。

首先，视频游戏是一种超高性能计算机程序。这意味着您将不能在时间关键代码部分和内存关键代码部分使用高级 API。从很大程度上说，您必须自己编写与游戏代码内部循环有关的一切内容，否则您的游戏将无法达到所需要的速度和性能。显然，这并不是说您不能信任像 DirectX 之类的 API，因为 DirectX 是作为一种高性能 API 编写的，并且进行了最大限度的精简。但是一般来说，您应该避免调用基于 Win32 API 的高级函数。例如，您可能认为 memset() 函数的速度相当快，但它只使用单字节进行填充。更好的填充方法是使用 QUAD（四元数），因为它可以一次写入 4 字节。例如，下面是我编写的用于 4 字节填充的内嵌汇编语言函数：

```
inline void Mem_Set_QUAD(void *dest, UINT data, int count)
{
// 这个函数填充 32 位对齐内存
// count 为 QUAD 数目

_asm
  {
  mov edi, dest  ; edi 指向目标内存
  mov ecx, count ; 要移动的 32 位字数
  mov eax, data  ; 32 位数据
  rep stosd      ; 移动数据
  } // end asm

} // end Mem_Set_QUAD
```

下面是使用 WORD（2 字节数据）进行填充的函数：

```
inline void Mem_Set_WORD(void *dest, USHORT data, int count)
{
// 这个函数填充 16 位对齐内存
// count 为 WORD 数目

_asm
  {
  mov edi, dest  ; edi 指向目标内存
  mov ecx, count ; 要移动的 16 位字数
  mov ax, data   ; 16 位数据
  rep stosw      ; 移动数据
  } // end asm

} // end Mem_Set_WORD
```

在有些情况下，这几行代码可以使游戏的帧频提高 2～4 倍！因此，当您考虑使用 API 调用时，需要清楚该 API 调用执行了哪些操作。

注意：奔腾 III、4 或更高级别的 CPU 支持 SIMD（Single Instruction Multiple Data，单指令多数据）指令，使得能够并行执行简单数学运算，因此优化基本数学运算的空间，如向量和矩阵计算，这将在后面介绍。

下面来看一看进行游戏编程时需要记住的一些技巧。

- 对您所做的工作进行备份。编写游戏代码时，很容易导致系统锁死。重新执行编写排序算法是一件事情，而重新编写角色的 AI 和冲突检测则是另一件事情。
- 开始游戏项目时，需要进行良好的组织。在项目中使用合理的文件名称和目录名称，使用一致的变量命名规则，并尽可能使用单独的目录存储图形和声音数据，而不要将所有内容存放到一个目录中。
- 使用内嵌函数。可以使用编译指令 inline 来完全避免函数调用。编译指令 inline 将使编译器尽可能

者看到的，状态切换是一个序列化过程。

1.3　通用游戏编程指南

下面介绍通用游戏编程技术和思想，您应该学习并尽可能采用这些技术和思想，从而更容易地进行游戏编程。

首先，视频游戏是一种超高性能计算机程序。这意味着您将不能在时间关键代码部分和内存关键代码部分使用高级 API。从很大程度上说，您必须自己编写与游戏代码内部循环有关的一切内容，否则您的游戏将无法达到所需要的速度和性能。显然，这并不是说您不能信任像 DirectX 之类的 API，因为 DirectX 是作为一种高性能 API 编写的，并且进行了最大限度的精简。但是一般来说，您应该避免调用基于 Win32 API 的高级函数。例如，您可能认为 memset() 函数的速度相当快，但它只使用单字节进行填充。更好的填充方法是使用 QUAD（四元数），因为它可以一次写入 4 字节。例如，下面是我编写的用于 4 字节填充的内嵌汇编语言函数：

```
inline void Mem_Set_QUAD(void *dest, UINT data, int count)
{
// 这个函数填充 32 位对齐内存
// count 为 QUAD 数目

_asm
  {
  mov edi, dest  ; edi 指向目标内存
  mov ecx, count ; 要移动的 32 位字数
  mov eax, data  ; 32 位数据
  rep stosd      ; 移动数据
  } // end asm

} // end Mem_Set_QUAD
```

下面是使用 WORD（2 字节数据）进行填充的函数：

```
inline void Mem_Set_WORD(void *dest, USHORT data, int count)
{
// 这个函数填充 16 位对齐内存
// count 为 WORD 数目

_asm
  {
  mov edi, dest  ; edi 指向目标内存
  mov ecx, count ; 要移动的 16 位字数
  mov ax, data   ; 16 位数据
  rep stosw      ; 移动数据
  } // end asm

} // end Mem_Set_WORD
```

在有些情况下，这几行代码可以使游戏的帧频提高 2～4 倍！因此，当您考虑使用 API 调用时，需要清楚该 API 调用执行了哪些操作。

注意：奔腾 III、4 或更高级别的 CPU 支持 SIMD（Single Instruction Multiple Data，单指令多数据）指令，使得能够并行执行简单数学运算，因此优化基本数学运算的空间，如向量和矩阵计算，这将在后面介绍。

下面来看一看进行游戏编程时需要记住的一些技巧。

- 对您所做的工作进行备份。编写游戏代码时，很容易导致系统锁死。重新执行编写排序算法是一件事情，而重新编写角色的 AI 和冲突检测则是另一件事情。
- 开始游戏项目时，需要进行良好的组织。在项目中使用合理的文件名称和目录名称，使用一致的变量命名规则，并尽可能使用单独的目录存储图形和声音数据，而不要将所有内容存放到一个目录中。
- 使用内嵌函数。可以使用编译指令 inline 来完全避免函数调用。编译指令 inline 将使编译器尽可能

者看到的，状态切换是一个序列化过程。

1.3　通用游戏编程指南

下面介绍通用游戏编程技术和思想，您应该学习并尽可能采用这些技术和思想，从而更容易地进行游戏编程。

首先，视频游戏是一种超高性能计算机程序。这意味着您将不能在时间关键代码部分和内存关键代码部分使用高级 API。从很大程度上说，您必须自己编写与游戏代码内部循环有关的一切内容，否则您的游戏将无法达到所需要的速度和性能。显然，这并不是说您不能信任像 DirectX 之类的 API，因为 DirectX 是作为一种高性能 API 编写的，并且进行了最大限度的精简。但是一般来说，您应该避免调用基于 Win32 API 的高级函数。例如，您可能认为 memset() 函数的速度相当快，但它只使用单字节进行填充。更好的填充方法是使用 QUAD（四元数），因为它可以一次写入 4 字节。例如，下面是我编写的用于 4 字节填充的内嵌汇编语言函数：

```
inline void Mem_Set_QUAD(void *dest, UINT data, int count)
{
// 这个函数填充 32 位对齐内存
// count 为 QUAD 数目

_asm
  {
  mov edi, dest  ; edi 指向目标内存
  mov ecx, count ; 要移动的 32 位字数
  mov eax, data  ; 32 位数据
  rep stosd      ; 移动数据
  } // end asm

} // end Mem_Set_QUAD
```

下面是使用 WORD（2 字节数据）进行填充的函数：

```
inline void Mem_Set_WORD(void *dest, USHORT data, int count)
{
// 这个函数填充 16 位对齐内存
// count 为 WORD 数目

_asm
  {
  mov edi, dest  ; edi 指向目标内存
  mov ecx, count ; 要移动的 16 位字数
  mov ax, data   ; 16 位数据
  rep stosw      ; 移动数据
  } // end asm

} // end Mem_Set_WORD
```

在有些情况下，这几行代码可以使游戏的帧频提高 2～4 倍！因此，当您考虑使用 API 调用时，需要清楚该 API 调用执行了哪些操作。

注意：奔腾 III、4 或更高级别的 CPU 支持 SIMD（Single Instruction Multiple Data，单指令多数据）指令，使得能够并行执行简单数学运算，因此优化基本数学运算的空间，如向量和矩阵计算，这将在后面介绍。

下面来看一看进行游戏编程时需要记住的一些技巧。

- 对您所做的工作进行备份。编写游戏代码时，很容易导致系统锁死。重新执行编写排序算法是一件事情，而重新编写角色的 AI 和冲突检测则是另一件事情。
- 开始游戏项目时，需要进行良好的组织。在项目中使用合理的文件名称和目录名称，使用一致的变量命名规则，并尽可能使用单独的目录存储图形和声音数据，而不要将所有内容存放到一个目录中。
- 使用内嵌函数。可以使用编译指令 inline 来完全避免函数调用。编译指令 inline 将使编译器尽可能

地把源代码加入到函数调用的位置，从而不必进行实际的函数调用。当然，这将会使程序变大，但是程序的速度更为重要。下面是一个范例：

```
inline void Plot_I(int x, int y, int color)
{
// 在屏幕上绘制一个像素
video_buffer[x + y*MEMORY_PITCH] = color;
} // end Plot_I
```

这里没有使用全局变量，因为编译器将执行相同类型的数据别名处理（data aliasing）。但是，如果在函数调用之间只有一两个参数发生改变，则使用全局变量将非常方便，因为未改变的变量值将可以继续使用，而不需要重新加载。

● 尽可能使用 32 位变量而不是 8 位或 16 位变量。奔腾以上的 CPU 都是 32 位 CPU，而安腾是 64 位的 CPU。这意味着它们不擅长处理 8 位或 16 位数据。实际上，更小的数据将会由于高速缓存和其他相关的内容寻址问题，降低程序的速度。例如，您可能创建一个类似如下的结构：

```
struct CPOINT
{
short x,y;
unsigned char c;
} // end CPOINT
```

尽管这个结构看起来不错，但实际上并非如此！首先，该结构本身的长度为 5 个字节（(2 * sizeof(short) +sizeof (char)) = 5）。这是相当糟糕的，将导致内存寻址的性能相当差。更好的方法是使用如下结构：

```
struct CPOINT
{
int x,y;
int c;
} // end CPOINT
```

注意：C++语言中的结构与类很相似，只是其默认可见性为 PUBLIC。

这个新结构要好得多。首先，它所有的元素大小相同——即 sizeof (int) = 4 字节。因而，可以使用一个指针在 DWORD 边界上递增，访问它的任何成员。当然，现在新结构的大小为（3 * sizeof (int)）=12 字节，但是它是 4 的倍数（即与 DWORD 边界对齐）；这将可以在相当大程度上改善程序的性能。

实际上，要想事情更完善些，可以填充所有结构，使得它们的尺寸为 32 字节的倍数。这是最优长度，因为在奔腾类处理器中，标准芯片级高速缓存线为 32 字节。可以手工进行填充，也可以使用编译器指令自动进行填充（这种方法更容易）。当然，这样可能会浪费一些内存，但是相对于所获得的速度而言它是值得的。

● 不要害怕使用全局变量。许多视频游戏在对于时间要求很严格的函数中都不使用参数，而是简单地使用全局参数。例如，对于如下函数：

```
void Plot(int x, int y, int color)
{
// 在屏幕上绘制一个像素
video_buffer[x + y*MEMORY_PITCH] = color;
} // end Plot
```

函数体所花费的时间比函数调用还要少！这是由于参数进栈和出栈而引起的。在这种情况下，更好的方法是先创建一个全局参数，并在调用函数前给它赋值：

```
int gx,gy,gz,gcolor; // 声明一些全局变量

void Plot_G(void)
{
// 根据全局变量来绘制像素
video_buffer[gx + gy*MEMORY_PITCH] = gcolor;

} // end Plot_G
```

● 以 RISC（Reduced Instruction Set Computer，精简指令集计算机）方式进行编程。使得您的代码尽

可能简单。奔腾类处理器特别适合处理简单指令，而不太擅长处理复杂指令。另外，通过使用简单指令使代码变得更长，将会使编译器更加容易处理。例如，不要编写类似如下的代码：

```
if (x+=(2*buffer[index++])>10)
{
// 执行处理
} // end if
```

而应该修改成如下代码：

```
x+=(2*buffer[index]);
index++;

if (x > 10)
{
// 执行处理
} // end if
```

使用后一种编码方法的原因有两个。首先，它允许调试器在代码段之间设置断点；其次，它使编译器更容易地为奔腾系列 CPU 简化代码，从而允许它使用多个执行单元并行处理更多代码。复杂的代码是相当糟糕的！

- 对于整数乘以 2 的幂这种简单运算，应使用二进制移位。因为所有数据在计算机中都是以二进制形式存储的，所以将数据向左移位或向右移位分别对应于乘法和除法。例如：

```
int y_pos = 10;

// 将 y_pos 乘以 64
y_pos = (y_pos << 6); // 2^6 = 64

// 将 y_pos 除以 8
y_pos = (y_pos >> 3); // 1/2^3 = 1/8
```

当然，在当今时代（尤其是在奔腾 4 上），有一些单循环乘法，但是移位将总是计算乘以 2 或除以 2 时最快的方法。

- 编写高效的算法。任何汇编语言也不能使 O（$n2$）算法变快。最好是使用干净、高效的算法。
- 不要在编写代码过程中优化代码。在编写游戏代码时，经常会忍不住去进行优化。这通常都是在浪费时间。等到已经完成一个主要代码块或已经完成游戏编码后，再进行仔细的优化（Intel 的 VTUNE 是一个很好的优化工具）。这将会节省大量时间，因为您不需要处理那些模糊的代码或不必要的优化。完成游戏编码之后，才是真正开始进行优化工作的时候。另一方面，不要编写不清晰的代码。
- 慎用 C++。如果您是一名经验丰富的专家，可以按所想的去做，但是不要过度地使用类或者重载一切东西。最后，简单、直观的代码是最好的代码，并且最容易进行调试。我从来不想看到多重继承！如果对于 Doom 和 Quake，C 语言已经足够了，则没有理由去使用 C++。如果它有所帮助，那很好；如果没有帮助，则不要在游戏编程中使用它，直到您非常熟悉游戏编程和 C++。
- 如果您发现所采取的路线很困难，则应该马上停止，然后备份，并想办法绕过该困难。我曾经见过许多程序员因为采用了糟糕的编程路线而最终葬送了他们的前程。您应该意识到，犯一个错误并重新编写 500 行代码，要比采用一个不好的编码结构强得多。因此，如果您在编程过程中发现了问题，应重新进行评价，并确认您所节省的时间是值得的。
- 对于简单对象，不要编写复杂的数据结构。不能因为链表功能强大，而在知道元素为 256 个的情况下，不使用数组而使用链表。在这种情况下，只需静态地分配数组，然后直接使用它。视频游戏编程中 90%的工作是数据操作。保持数据尽可能简单并可见，以便可以迅速访问它并执行所需的操作。确保数据结构与需要解决的问题相称。

1.4　使 用 工 具

在过去，编写视频游戏的工具无非是文本编辑器以及自编的绘图和声音程序。但是在今天，事情要稍

微复杂些。要编写 3D 游戏，至少需要 C/C++编译器、2D 绘图程序、声音处理程序以及某种 3D 建模程序（除非您想要以 ASCII 格式输入所有 3D 模型数据）。另外，如果要使用任何 MIDI 音乐，还将需要一个音乐序列化程序。

下面来看一些比较流行的产品以及它们的功能。

- C/C++编译器：对于 Windows 9x/Me/XP/2000/NT 开发，最好的编译器当然是 MS VC++，如图 1.3 所示。它可以做您需要它做的一切事情。它所生成的.EXE 程序是执行速度最快的代码。Borland 公司的编译器功能也很强大（且价格更便宜一些），但它的功能相对来说要少一些。不管哪个编译器，您都不需要其全功能的版本——能生成 Win32 .EXE 文件的学习版足够了。



图 1.3　Microsoft VC++环境

- 2D 位图绘图软件：需要一个绘图程序，用于绘图或图像处理。绘图程序主要用于让您按像素来绘制图像并对图像进行操作。据我所知，JASC 公司的 Paint Shop Pro 软件（如图 1.4 所示）的性价比最高。但大多数人最喜欢的是 Adobe Photoshop，其功能更强大。

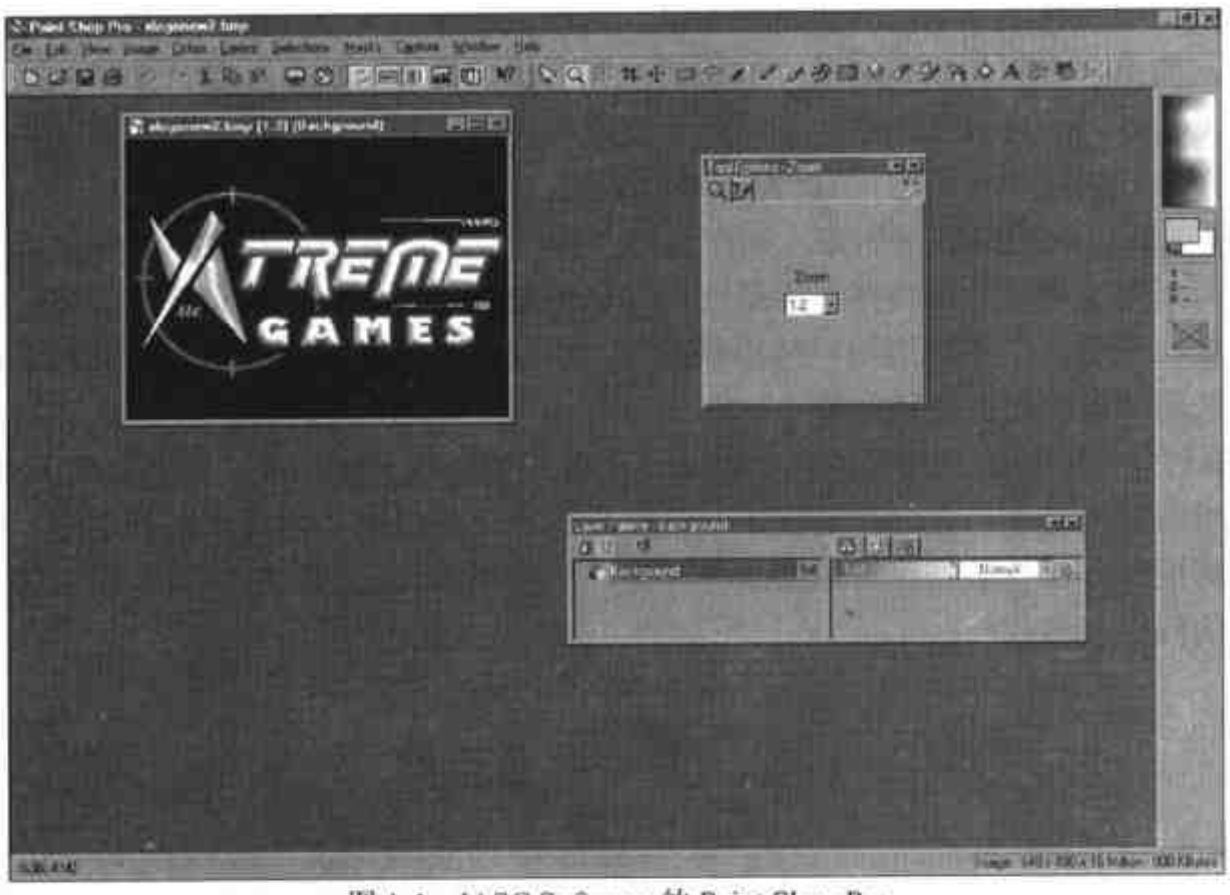


图 1.4　JASC Software 的 Paint Shop Pro

● 2D 矢量绘图软件：另一方面，绘图程序允许用曲线、直线和其他 2D 几何图形构造图像。这种类型的程序不是必需的，但如果需要，Adobe Illustrator 是很好的选择。

● 图像后处理：图像处理软件是您所需的最后一种 2D 绘图程序。这些程序大多用于进行图像后处理，而很少用于创建图像。在这一方面，大多数人都喜欢选择 Adobe Photoshop，但是我认为 Corel Photo Paint 更好——哪一个更好，需要您自己决定。

● 声音处理软件：在当今游戏使用的声音 FX 中，90%以上都是数字取样的。要对这种类型的声音数据进行处理，需要一个数字声音处理程序。在这方面，Sound Forge（如图 1.5 所示）是最好的程序之一。它是迄今为止我所见过的最复杂的声音处理程序之一，也是使用起来最简单的。我现在还在不断发现它的新功能！Cool Edit Pro 的功能也非常强大，但我很少使用它。

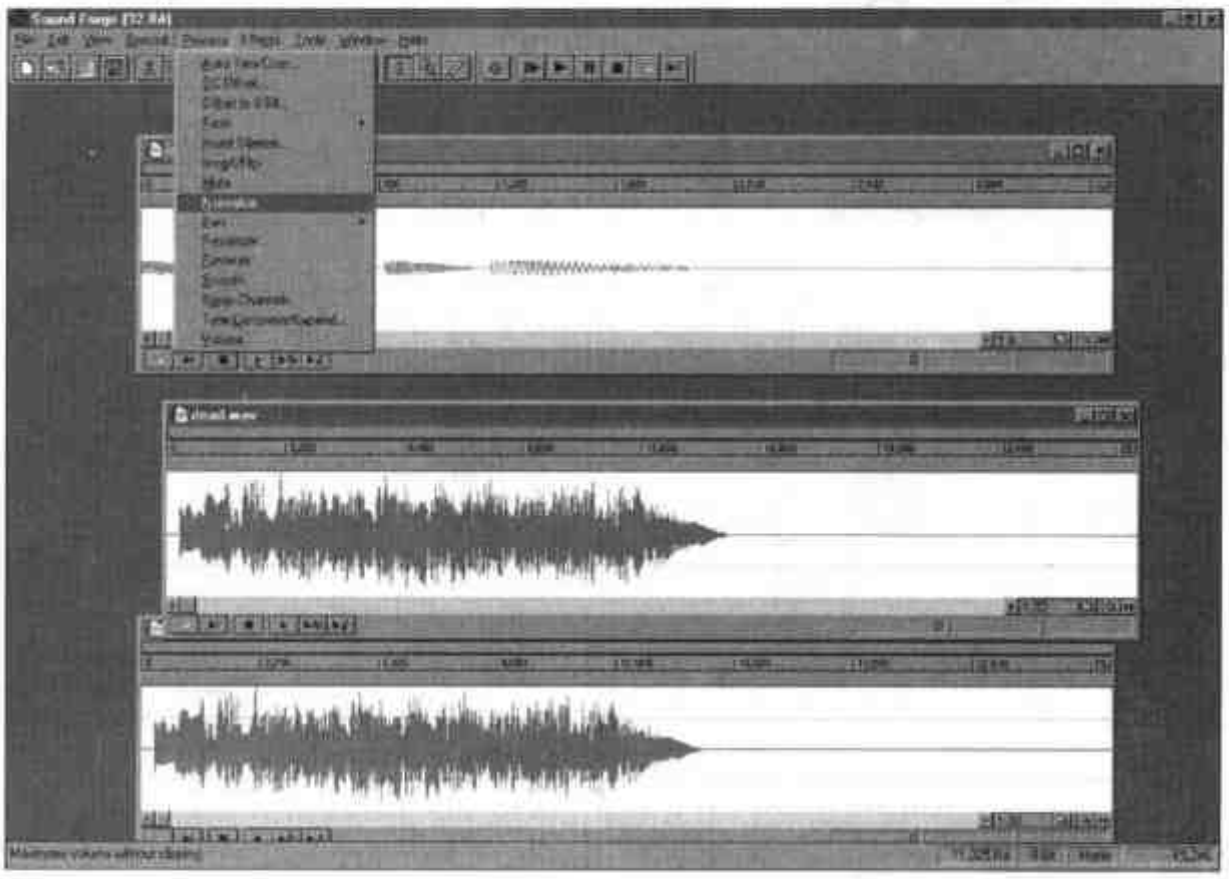


图 1.5　运行中的 Sound Forge

● 3D 建模程序：3D 建模程序的选择是最具挑战性的。3D 建模程序可能需要花费数万美元，但是近来也出现了很多价格便宜的建模程序，它们也有创建动画所需的功能。对于简单到中等规模的 3D 模型和动画，我主要使用一个名为 Caligari trueSpace 的建模程序，如图 1.6 所示。我认为它是性价比最高的建模程序——只需要几百美元，同时具有最好的界面（在我看来）。

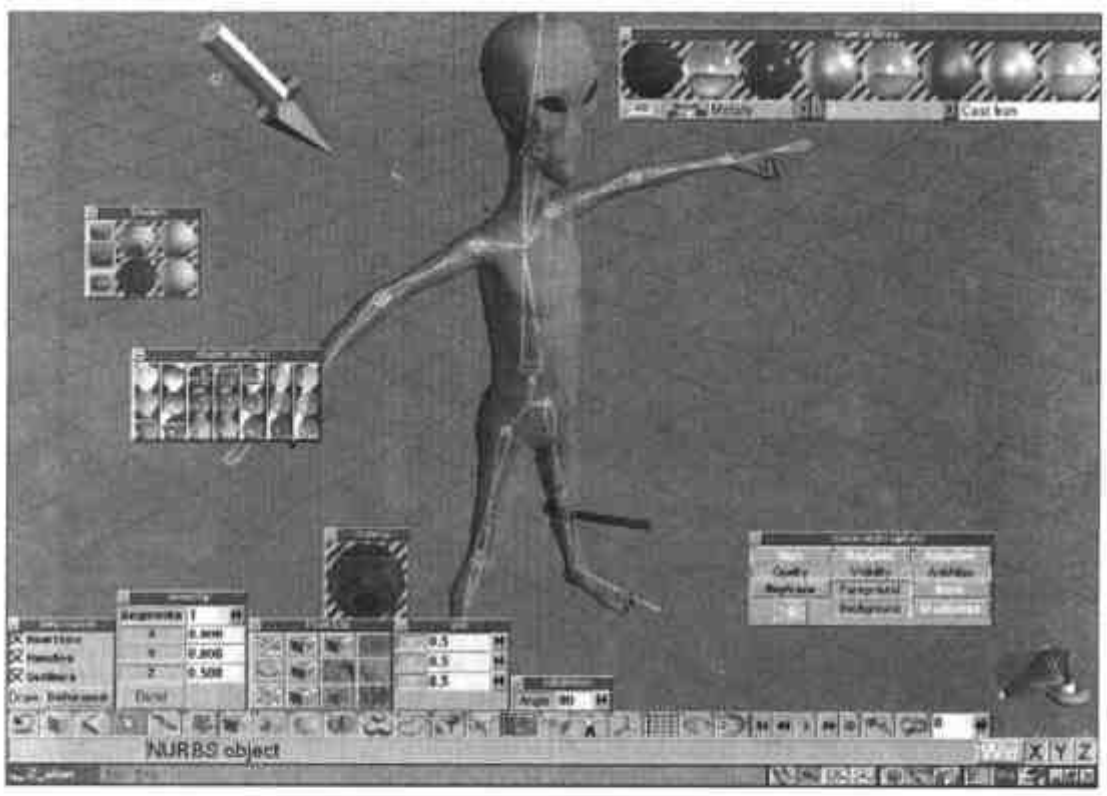


图 1.6　Caligari 的 trueSpace 3D 建模程序

如果您需要更多的功能和接近于照片质量的真实感，可以考虑选择 3D Studio Max（如图 1.7 所示），但它需要 2500 美元甚至更多。然而，由于我们大多数时候都仅使用建模程序来创建 3D 网格，而不进行渲染，因此 trueSpace 是正确的选择，也可以考虑选择免费或共享的建模程序。

注意：在 Internet 上有很多免费或共享的 3D 建模程序，在有些情况下它们的功能与商业 3D 建模软件的功能一样强大。因此，如果您存在资金上的困难，可考虑到 Internet 上搜寻一些免费或共享的 3D 建模程序。

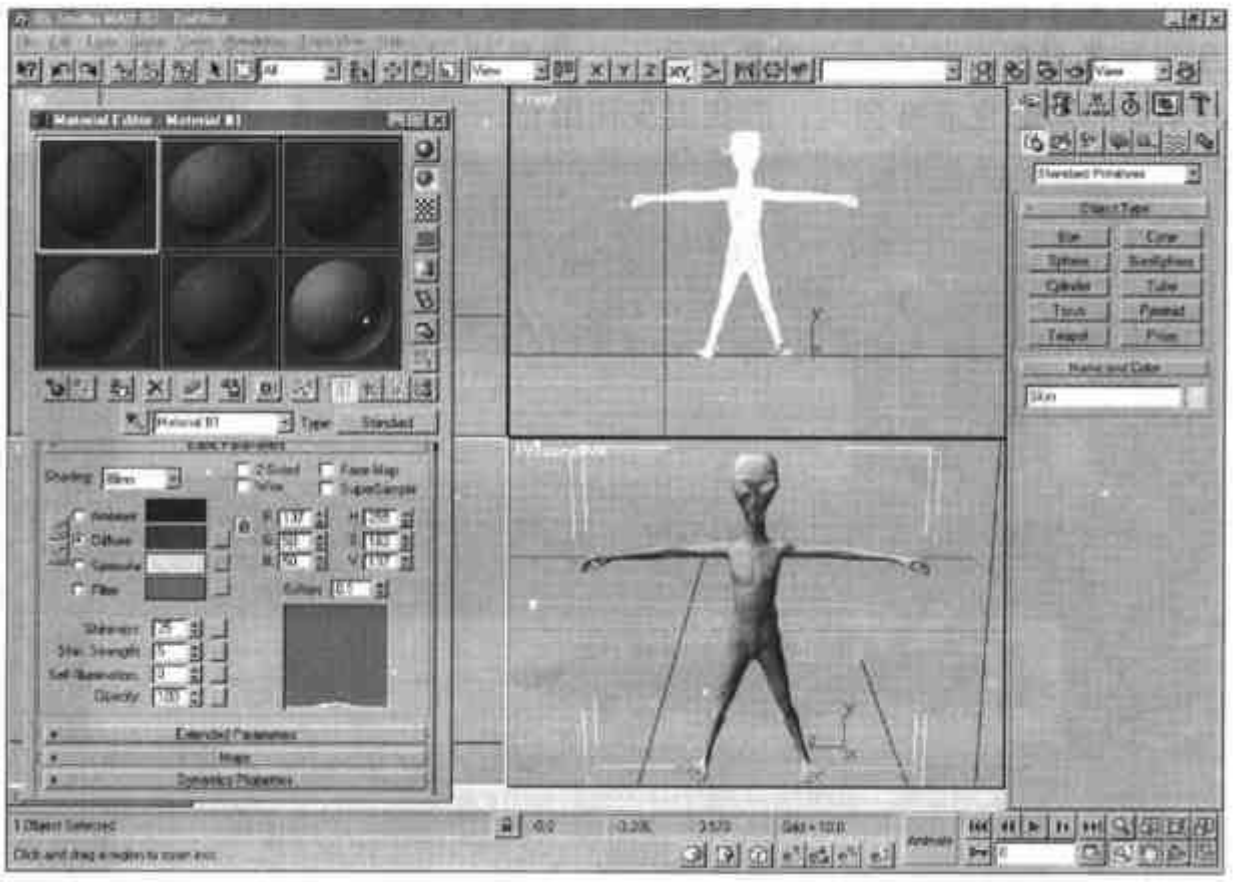


图 1.7　3D Studio Max

1.4.1　3D 关卡编辑器

本书将编写一个基于软件的 3D 引擎，但不打算编写复杂的工具来创建 3D 室内模型。当然可以用 3D 建模程序来创建 3D 世界，但有更好的程序来做这些事情，如 WorldCraft，如图 1.8 所示。因而，在编写 3D 游戏和引擎时最好的方法是使用一种与其他常用编辑器（如 WorldCraft 或类似的编辑器）输出类型兼容的文件格式和数据结构，这样将可以使用他人的工具来创建自己的游戏世界。当然，这些编辑器的文件格式大多数以 id Software 的工作和 Quake 引擎为基础，但是不管是否基于 Quake 文件格式，它应该是一种已被广泛采用的格式。

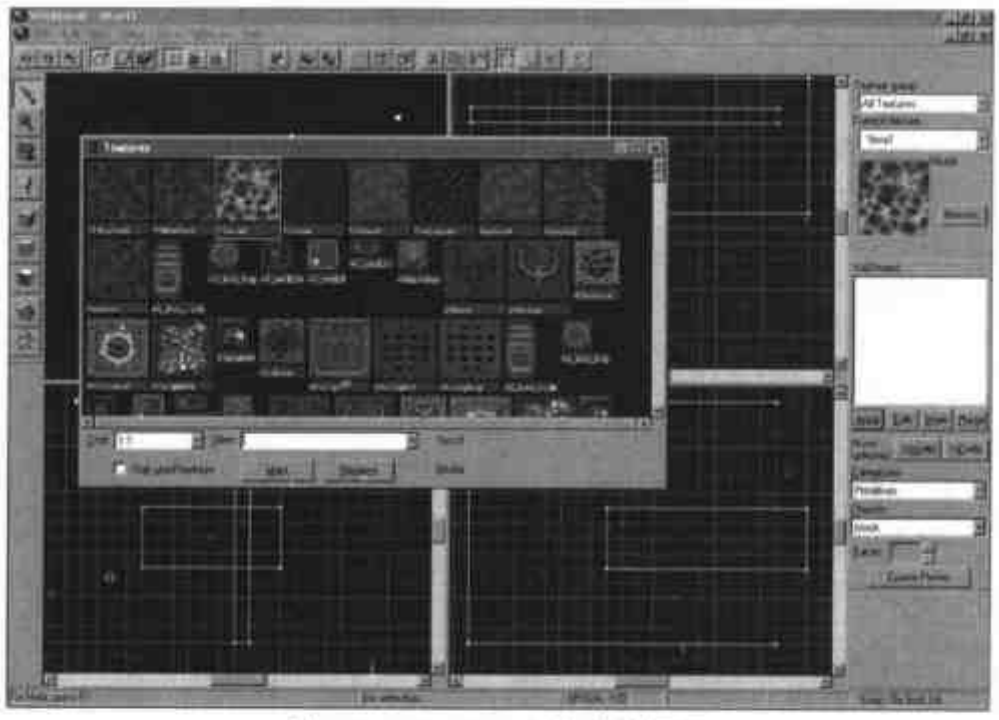


图 1.8　WorldCraft 关卡编辑器

● 音乐和 MIDI 序列化程序：在当今的游戏中主要有两种类型的音乐——纯数字音乐（如 CD）和 MIDI（音乐设备数字接口）音乐，其中 MIDI 是根据音节数据合成的。要操作 MIDI 信息和歌曲，需要一个序列化软件。在这方面最好的软件之一是 CakeWalk，如图 1.9 所示。CakeWalk 的价格也比较合理，因此如果您计划录制或操作任何 MIDI 音乐，建议考虑该软件。介绍 DirectMusic 时，将进一步介绍 MIDI 数据。

注意：前面所介绍的许多软件的出版商允许我将其软件的共享版本和评估版本放到本书的光盘中，因此您可以在本书的光盘中找到它们。

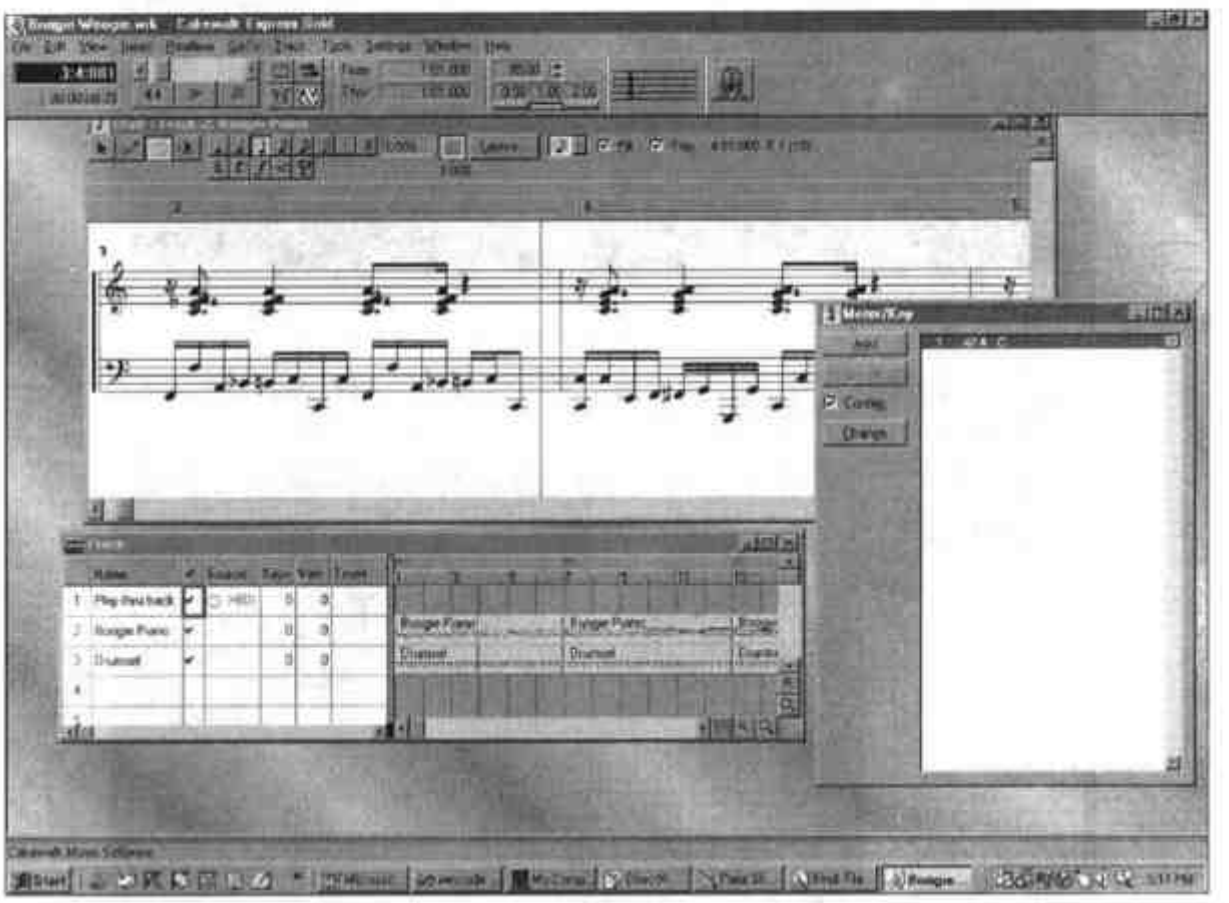


图 1.9　CakeWalk MIDI 音轨序列化程序

1.4.2　使用编译器

学习 3D Windows 游戏编程最困难的地方之一是学习如何使用编译器。在大多数情况下，人们会兴奋地急于开始，急着用 IDE 来编译，而编译时无数的编译错误和链接错误便冒了出来！为帮助解决这些问题，这里介绍几个基本的编译器概念。

注意：这里介绍的细节都与 Visual C++编译器相关，但其思想适用于任何编译器。

1．阅读整个编译器手册。

2．必须在系统中安装 DirectX 9.0 SDK。为此，可打开附带光盘中的 DirectX SDK 目录，阅读其中的文件 README.TXT，并根据其中的说明执行安装程序——双击 DirectX SDK 安装程序。但是别忘了，本书使用 DirectX 9.0（但只使用了 DirectX 7/8 的接口），因此您需要使用附带光盘中的 DirectX SDK。如果愿意，也可以使用 DirectX 8.0 编译所有程序。

3．除非特别说明，我们将生成 Win32.EXE 应用程序，而不是.DLL、ActiveX 组件、控制台应用程序等。因此使用编译器时，首先要创建一个新工程，并将目标输出文件设置为 Win32.EXE。图 1.10 说明了如何在 Visual C++ 6.0 编译器中完成这种设置。

注意：编译时可以建立两种类型的。EXE 程序——Release（发布版）和 Debug（调试版）。发布版经过了优化、执行速度较快；而调试版保存了调试信息、执行速度较慢。建议开发时建立调试版程序，直到它能够正常工作时，再将编译器设置成发布版模式。

4．从主菜单中选择 Add Files 命令，将源文件加入到工程中。图 1.11 说明了如何在 Visual C++ 6.0 编译器中执行这一步。

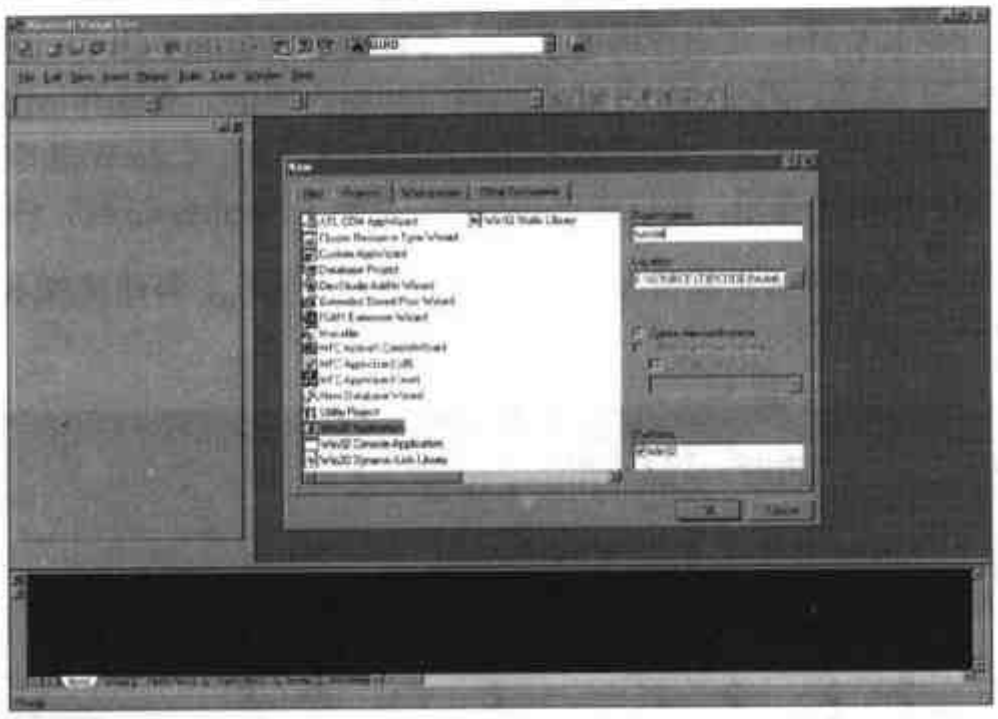


图 1.10　在 Visual C++ 6.0 中创建 Win32.EXE 程序

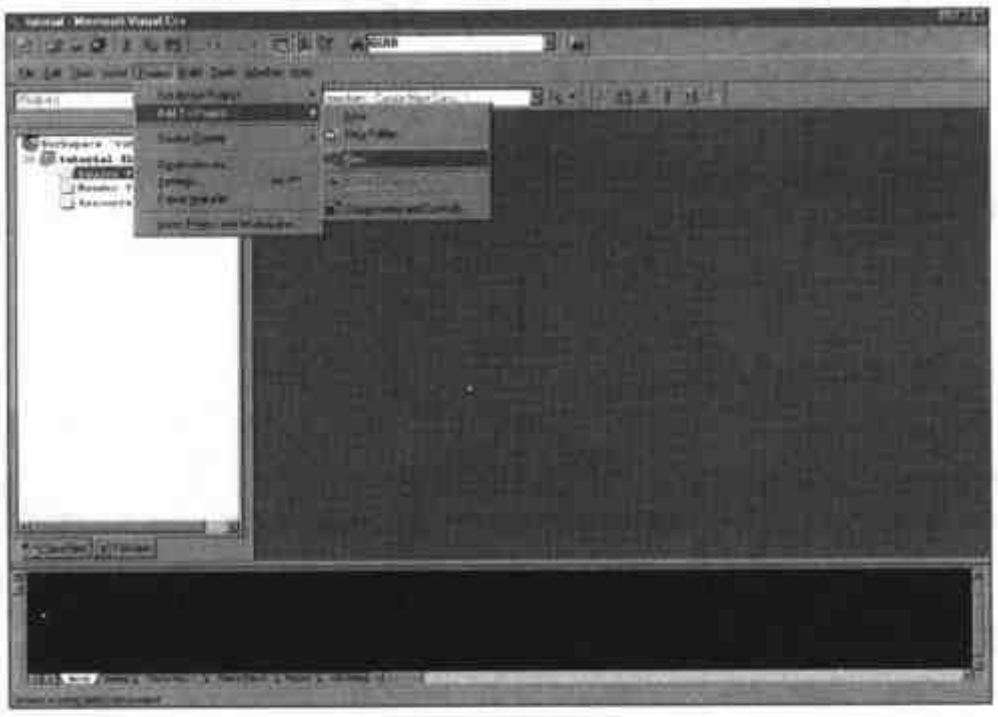


图 1.11　在 Visual C++ 6.0 中将文件加入到工程中

5. 编译任何需要 DirectX 的程序时，需要在编译器的搜索路径中包含 DirectX 头文件和 DirectX .LIB 文件的位置。为此，可以在 IDE 的主菜单中选择 Options/Directories，然后将合适的路径加入到变量 Include Files 和 Library Files 中，如图 1.12 所示。

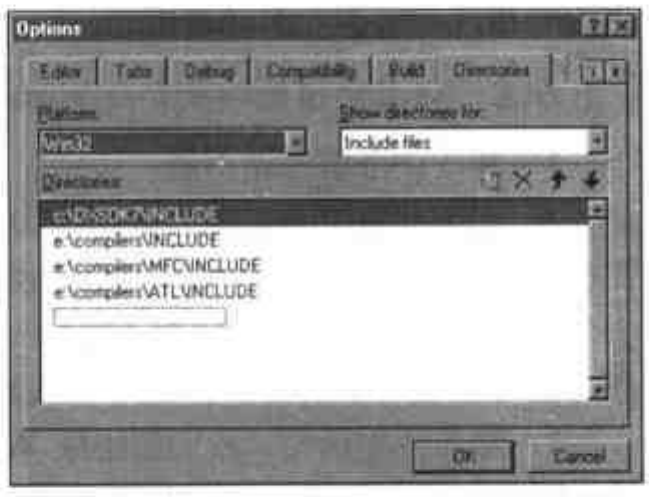


图 1.12　在 Visual C++ 6.0 中设置搜索路径

注意：确保 DirectX 搜索节点位于列表中的第一位置。编译器可能查找老版本的 DirectX，而您并不想使用它。

6. 另外，还必须在工程的链表或项目资源列表中包含 DirectX COM 接口导入库，如图 1.13 所示。

- DDRAW.LIB；
- DSOUND.LIB；
- DINPUT.LIB；
- DINPUT8.LIB；
- 特定范例所需的其他库文件。

这些 DirectX .LIB 文件位于 DirectX SDK 安装目录下的 LIB\目录中。必须将这些.LIB 文件加入到工程中，而不能仅仅将目录 LIB\加入到搜索路径中——因为它只告诉编译器在哪里进行搜索，而不会让编译器/链接器实际链接这些 DirectX .LIB 文件。我收到过无数人的电子邮件，他们因为没有执行这一步而遇到了问题。可以选择菜单 Project/Settings，然后在 Link 选项卡中的 Object/Library-Modules 文本框中添加这些文件，如图 1.14 所示。

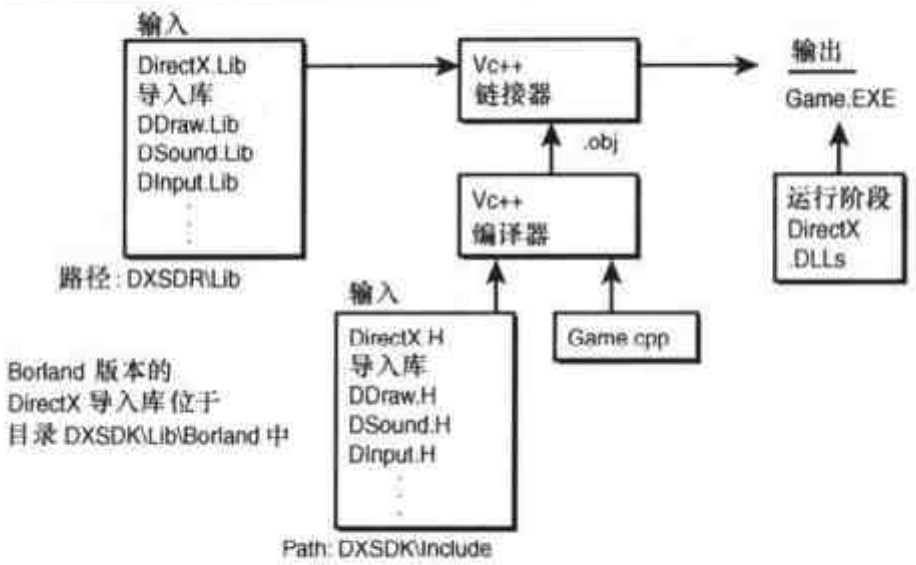


图 1.13　创建 Win32 DirectX 应用程序所需的资源

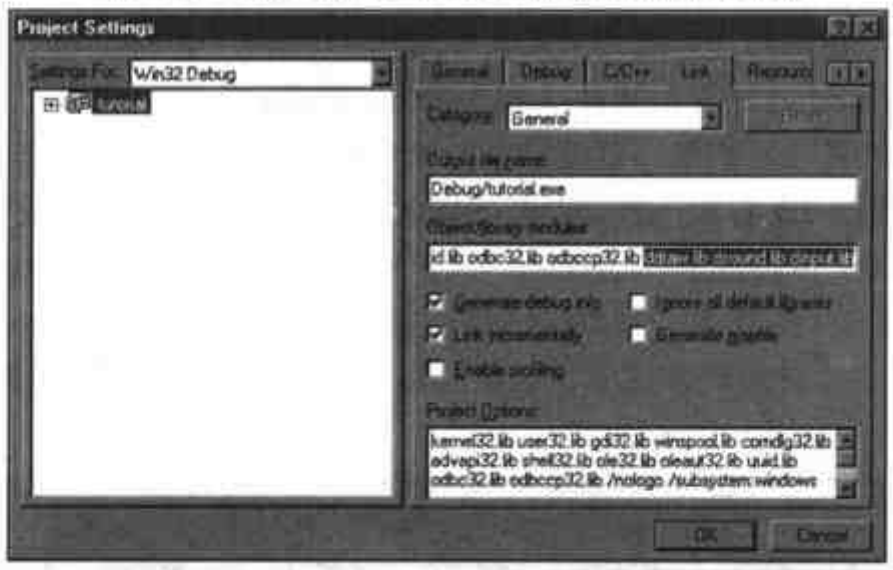


图 1.14　在 Visual C++ 6.0 中将 DirectX .LIB 文件加入到链接列表中

注意：如果您使用的是 Visual C++，可能还需将 Windows 多媒体扩展库 WINMM.LIB 加入到工程中。该文件位于 Visual C++编译器安装位置的 LIB\目录下。如果在这里找不到，可以使用菜单“开始/查找”来搜索，然后将它加入到链接列表中。

7. 现在编译器已经准备好了。

注意：如果您是 Borland 用户，在 DirectX SDK 中有一个单独的 BORLAND\库目录，应加入其中的.LIB 文件，而不是 Visual C++ DirectX .LIB 文件。

如果现在您仍对此有疑问，也不要着急——本书中编译程序时，将重温这些步骤。

注意：您肯定听说过新的 Visual .NET 系统。从技术上说，它是最新的 Microsoft 编译器和开发技术，是一种崭新的技术。它只能在 Windows XP/2000 平台上运行，对我们的目的来说功能太强，因此我们仍使用 Visual C++ 6.0。但是，一切内容也都适用于.NET：Win32 就是 Win32!

1.5　一个 3D 游戏范例：*Raiders 3D*

介绍数学、3D 游戏编程和图形学之前，先介绍一个完整的 3D 太空游戏——虽然很简单，但也是游戏。这样您将可以看到真实的游戏循环、图形调用和编译过程。

问题在于，现在是第 1 章，因此不能使用后续章节将介绍的技术。同时，本章还没有介绍《Windows 游戏编程大师技巧》中的 API 引擎，因而我决定让您习惯于在游戏编程中使用“黑盒”API。该范例程序将让您习惯于使用 API（如 DirectX 的 API）和《Windows 游戏编程大师技巧》中的游戏引擎。

注意：尽管在 20 世纪的 70 年代、80 年代，甚至 90 年代早期，可以自己动手完成一切内容，但在 21 世纪，计算机中有太多的硬件和软件子系统，不太可能自己编写一切内容。作为游戏编程人员，我们只能屈服，使用一些 API，如 Win32、DirectX 等。

根据“黑盒”需求，创建一个 16 位 3D 太空游戏至少需要哪些东西呢？

我们需要 API 提供的如下功能：

- 使用 DirectX 切换到任何图形模式；
- 在屏幕上绘制彩色线条和像素；
- 获得键盘输入；
- 播放磁盘上.WAV 文件中的声音；
- 播放磁盘上.MID 文件中的 MIDI 音乐；
- 使用定时函数同步游戏循环；
- 在屏幕上绘制彩色文本字符串；
- 复制后缓存或离屏渲染页到屏幕上。

所有这些功能都位于《Windows 游戏编程大师技巧》中创建的游戏库模块 T3DLIB*中，它由 6 个文件组成：T3DLIB1.CPP|H、T3DLIB2.CPP|H、T3DLIB3.CPP|H。模块 1 包含大部分 DirectX 功能，模块 2 包含大部分 DirectInput 和 DirectSound 功能，模块 3 包含 DirectMusic 功能。

使用 T3DLIB*游戏库的最小函数集，我编写了一个叫作 *Raiders 3D* 的游戏，以演示本章介绍的概念。另外，由于该游戏是一个线框 3D 游戏，因此很多代码读者可能看不懂。尽管我将简要介绍一些 3D 数学和算法，但不是要花太多时间去试图理解一切代码；该范例只是为了提供一个整体感觉。

Raiders 3D 说明了真实 3D 游戏的主要组成部分，包括游戏循环、计分（scoring）、AI、碰撞、声音和音乐。图 1.15 是该游戏在运行时的屏幕截图。当然，它不可与 Star Wars 同日而语，但相对于几小时的工作而言已经相当不错了！



图 1.15　Raiders 3D 的屏幕截图

介绍游戏的源代码之前，先来看一下该工程及其各个组成部分的组织结构，如图 1.16 所示。

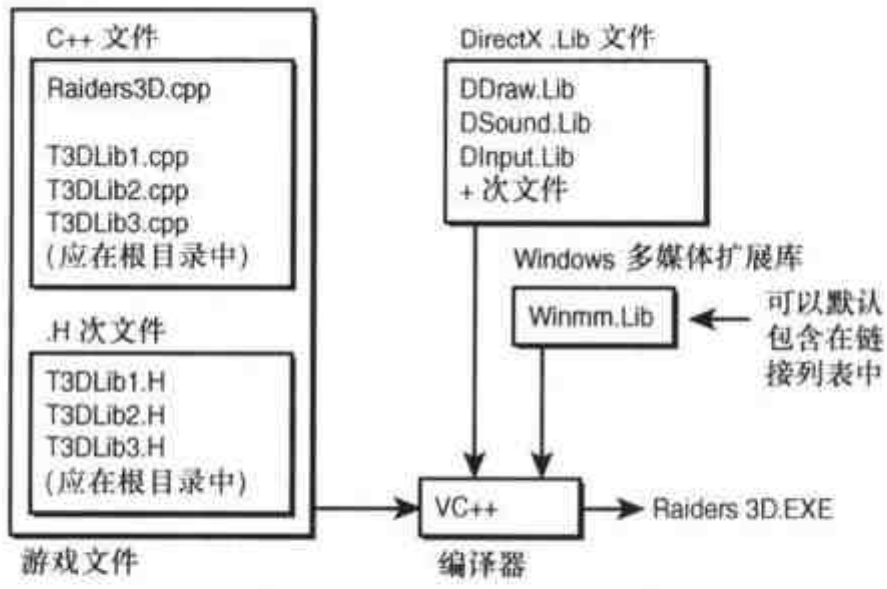


图 1.16　Raiders 3D 的代码结构

该游戏由下列文件组成。

- RAIDERS3D.CPP——主游戏循环，它使用 T3DLIB 的功能并创建一个最小的 Win32 应用程序。
- T3DLIB1.CPP、T3DLIB2.CPP、T3DLIB3.CPP——T3DLIB 游戏库源文件。
- T3DLIB1.H、T3DLIB2.H、T3DLIB3.H——游戏库的头文件。
- DDRAW.LIB——DirectDraw 是 DirectX 的 2D 图形组件。导入库 DDRAW.LIB 是建立应用程序所必需的。它没有包含实际的 DirecX 代码，而是一个中间库，您可以对它进行调用，它再加载动态链接库 DDRAW.DLL 来完成实际的工作。该文件位于 DirectX SDK 安装目录中的 LIB\目录下。
- DINPUT.LIB/DINPUT8.LIB——DirectInput 是 DirectX 的设备输入组件。导入库 DINPUT.LIB 和 DINPUT8.LIB 是建立应用程序所必需的。它们没有包含实际的 DirecX 代码，而是一个中间库，您可以调用它们，它们再加载动态链接库 DINPUT.DLL 和 DINPUT8.DLL 来完成实际的工作。这些文件位于 DirectX SDK 安装目录中的 LIB\目录下。
- DSOUND.LIB——DirectSound 是 DirectX 的数字声音组件。导入库 DSOUND.LIB 是建立应用程序所必需的。它没有包含实际的 DirecX 代码，而是一个中间库，您可以调用它，它再加载动态链接库 DSOUND.DLL 来完成实际的工作。该文件位于 DirectX SDK 安装目录中的 LIB\目录下。

注意：并不存在文件 DMUSIC.LIB，尽管 T3DLIB 也使用 DirectMusic。这是因为 DirectMusic 是一个纯粹的 COM（组件对象模型）。也就是说，不存在这样的导入库，即它封装对 DirectMusic 的各种函数调用，您需要自己完成这些工作。幸运的是，我已经帮您完成了这些工作！

下面这些文件不是编译器或链接器所需要的文件，而是 DirectX 运行阶段 DLL，游戏应用程序启动时需要加载它们。

- DINPUT.DLL/DINPUT8.DLL——运行阶段 DirectInput 库文件，包含了 DirectInput 接口函数的 COM 实现，应用程序通过 DINPUT.LIB 导入库来调用它们。您不需要关心这些动态链接库本身，只需要确认已经正确安装了 DirectX 运行阶段文件。
- DDRAW.DLL——运行阶段 DirectDraw 库文件，包含了 DirectDraw 接口函数的 COM 实现，应用程序通过 DDRAW.LIB 导入库来调用它们。您不需要关心这个动态链接库本身，只需确认已经正确安装了 DirectX 运行阶段文件。
- DSOUND.DLL——运行阶段 DirectSound 库文件，包含了 DirectSound 接口函数的 COM 实现，应用程序通过 DSOUND.LIB 导入库来调用它们。您不需要关心这个动态链接库本身，只需确认已经正确安装了 DirectX 运行阶段文件。
- DMUSIC.DLL——运行阶段 DirectMusic 库文件，包含了 DirectMusic 接口函数的 COM 实现，应用程序通过 COM 调用来直接调用它们。您不需要关心这个动态链接库本身，只需确认已经正确安装了 DirectX 运行阶段文件。

通过对库文件的几次调用，我创建了游戏 RAIDERS3D.CPP，如下面的程序清单所示。该游戏在 16 位颜色和窗口模式下运行，因此必须将桌面设置成 16 位颜色模式。

请仔细阅读该游戏的代码，尤其是主游戏循环、3D 数学运算以及对游戏处理函数的调用。

```
// RAIDERS3D.CPP--本书的第一个 3D 游戏
// 要编译该程序，请务必在工程的链接列表中包含
// DDRAW.LIB、DSOUND.LIB、DINPUT.LIB 和 WINMM.LIB
// 当然还有 C++源代码模块 T3DLIB1.CPP、T3DLIB2.CPP 和 T3DLIB3.CPP
// 另外，还需要将头文件 T3DLIB1.H、T3DLIB2.H 和 T3DLIB3.H 放在工作目录中
// 以便编译器能够找到它们

// 要运行该游戏，务必将桌面设置为 16 位颜色模式，分辨率为 640x480 或更高

// INCLUDES ///////////////////////////////////////////////

#define INITGUID     // 确保所有的 COM 接口可用
            // 也可以不这样做，而是包含.LIB 文件 DXGUID.LIB

#define WIN32_LEAN_AND_MEAN
```

```
#include <windows.h>   // 包含重要的Windows 头文件
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // 包含重要的 C/C++头文件
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h>  // 包含 directX 头文件
#include <dsound.h>
#include <dmksctrl.h>
#include <dmusici.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <dinput.h>
#include "T3DLIB1.h" // 包含游戏库模块头文件
#include "T3DLIB2.h"
#include "T3DLIB3.h"

// DEFINES ////////////////////////////////////////////////

// 用于Windows 接口的常量
#define WINDOW_CLASS_NAME "WIN3DCLASS"  // 类名
#define WINDOW_TITLE    "T3D Graphics Console Ver 2.0"
#define WINDOW_WIDTH    640  // 窗口大小
#define WINDOW_HEIGHT    480

#define WINDOW_BPP      16    // 窗口的位深（8、16、24 等）
                  // 注意：如果使用窗口、非全屏模式
                  // 位深必须与系统位深相同
                  // 另外，如果 8 位颜色模式，将创建一个调色板
                  // 并将其与应用程序关联起来

#define WINDOWED_APP      1      // 0 表示非窗口模式；1 表示窗口模式

//////////////////////////////////////////////////////////

#define NUM_STARS  512  // 星星数
#define NUM_TIES   32   // 飞船数

// 3D engine constants
#define NEAR_Z      10   // 近裁剪面
#define FAR_Z       2000 // 远裁剪面
#define VIEW_DISTANCE  320  // 视距
                // 投影到宽为 640 的窗口上时，视野将为 90 度
// 与玩家相关的常量
#define CROSS_VEL    8  // 十字准星的移动速度
#define PLAYER_Z_VEL  8  // 玩家沿 z 轴的虚拟速度
                // 用于在不移动的情况下模拟运动

// 与飞船模型相关的常量
#define NUM_TIE_VERTS    10
#define NUM_TIE_EDGES    8

// 与爆炸结构相关的常量
#define NUM_EXPLOSIONS   (NUM_TIES) // 爆炸结构总数

// 游戏状态
#define GAME_RUNNING     1
```

```
#define GAME_OVER      0

// 数据结构 ////////////////////////////////////////////////////////

// 3D 点
typedef struct POINT3D_TYP
    {
    USHORT color;    // 16 位颜色
    float x,y,z;    // 3D 坐标
    } POINT3D, *POINT3D_PTR;

// 3D 线段，由两个指向顶点列表的索引来定义线段的端点
typedef struct LINE3D_TYP
    {
    USHORT color;  // 线段的 16 位颜色
    int v1,v2;    // 定义线段端点的顶点列表索引

    } LINE3D, *LINE3D_PTR;

// TIE 飞船
typedef struct TIE_TYP
    {
    int state;     // 飞船的状态，0 表示死亡，1 表示活着
    float x, y, z; // 飞船的位置
    float xv,yv,zv; // 飞船的速度
    } TIE, *TIE_PTR;

// 用于表示速度的 3D 向量
typedef struct VEC3D_TYP
    {
    float x,y,z; // 向量的坐标
    } VEC3D, *VEC3D_PTR;

// 用于模拟爆炸的数据结构
typedef struct EXPL_TYP
    {
    int state;     // 爆炸状态
    int counter;    // 爆炸计数器
    USHORT color;   // 颜色

    // 爆炸是通过随机移动构成飞船 3D 模型的线段/边来实现的
    POINT3D p1[NUM_TIE_EDGES];  // 第 n 条边的起点
    POINT3D p2[NUM_TIE_EDGES];  // 第 n 条边的终点

    VEC3D   vel[NUM_TIE_EDGES]; // 炮弹碎片的速度

    } EXPL, *EXPL_PTR;

// 函数原型 ////////////////////////////////////////////////////////

// 游戏控制台函数
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// 游戏函数
void Init_Tie(int index);

// 全局变量 ////////////////////////////////////////////////////////

HWND main_window_handle      = NULL; // 用于存储窗口句柄
HINSTANCE main_instance      = NULL; // 用于存储实例
char buffer[256];                    // 用于打印文本

// 飞船是由连接顶点的线段组成的
// 只有一个飞船模型，所有飞船都是根据该模型变换得到的
```

```
POINT3D tie_vlist[NUM_TIE_VERTS]; // 飞船模型的顶点列表
LINE3D  tie_shape[NUM_TIE_EDGES]; // 飞船模型的边列表
TIE     ties[NUM_TIES];      // 飞船数组

POINT3D stars[NUM_STARS];  // 星空

// 一些颜色
USHORT rgb_green,
    rgb_white,
    rgb_red,
    rgb_blue;

// 与玩家相关的变量
float cross_x = 0, // 准星的位置
   cross_y = 0;

int cross_x_screen  = WINDOW_WIDTH/2,   // 准星坐标
  cross_y_screen  = WINDOW_HEIGHT/2,
  target_x_screen = WINDOW_WIDTH/2,   // 瞄准位置
  target_y_screen = WINDOW_HEIGHT/2;

int player_z_vel = 4; // 视点的虚拟速度
int cannon_state = 0; // 激光炮的状态
int cannon_count = 0; // 激光炮计数器

EXPL explosions[NUM_EXPLOSIONS]; // 爆炸结构数组

int misses = 0; // 记录逃脱的飞船数
int hits   = 0; // 记录被击落的飞船数
int score  = 0; // 请读者猜一猜这个变量的用途

// 音乐和声音
int main_track_id = -1, // 主音轨 ID
  laser_id     = -1, // 激光脉冲的声音
  explosion_id  = -1, // 爆炸声
  flyby_id     = -1; // 飞船飞过的声音

int game_state = GAME_RUNNING; // 游戏状态

// 函数 ////////////////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
                  UINT msg,
               WPARAM wparam,
               LPARAM lparam)
{
// 这是系统的主消息处理函数
PAINTSTRUCT  ps;   // 用于 WM_PAINT 中
HDC       hdc;   // 设备场境句柄

// 是什么消息?
switch(msg)
  {
  case WM_CREATE:
    {
    // 执行初始化工作
    return(0);
    } break;

  case WM_PAINT:
     {
     // 开始绘制
     hdc = BeginPaint(hwnd,&ps);

     // 结束绘制
     EndPaint(hwnd,&ps);
     return(0);
```

```
    } break;

  case WM_DESTROY:
    {
    // 删除应用程序
    PostQuitMessage(0);
    return(0);
    } break;

  default:break;

  } // end switch

// 处理其他消息
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////////

int WINAPI WinMain( HINSTANCE hinstance,
          HINSTANCE hprevinstance,
          LPSTR lpcmdline,
          int ncmdshow)
{
// 这是 winmain 函数

WNDCLASS winclass;  // 存储创建的类
HWND    hwnd;    // 窗口句柄
MSG    msg;    // 消息
HDC    hdc;    // 设备场境句柄
PAINTSTRUCT ps;   // paintstruct

// 设置窗口类
winclass.style    = CS_DBLCLKS | CS_OWNDC |
            CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc  = WindowProc;
winclass.cbClsExtra  = 0;
winclass.cbWndExtra  = 0;
winclass.hInstance  = hinstance;
winclass.hIcon    = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor  = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground    = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName    = NULL;
winclass.lpszClassName    = WINDOW_CLASS_NAME;

// 注册窗口类
if (!RegisterClass(&winclass))
  return(0);

// 创建窗口，根据 WINDOWED_APP 是否为 true 来选择合适的窗口属性
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // 类
       WINDOW_TITLE,    // 标题
 (WINDOWED_APP ? (WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION) : (WS_POPUP | WS_VISIBLE)),
       0,0,    //  x、y 坐标
       WINDOW_WIDTH,  // 宽度
       WINDOW_HEIGHT, // 高度
       NULL,     // 父窗口句柄
       NULL,     // 菜单句柄
       hinstance, // 实例
       NULL)))   // 创建参数
return(0);

// 将窗口句柄和实例赋给全局变量
main_window_handle = hwnd;
main_instance     = hinstance;
```

```
// 调整窗口大小，使客户区的大小为 width×height
if (WINDOWED_APP)
{
// 调整窗口大小，使客户区为实际请求的大小
// 因为窗口应用程序中包含大量的控件和边界
// 如果应用程序不是窗口模式，则无关紧要
RECT window_rect = {0,0,WINDOW_WIDTH-1,WINDOW_HEIGHT-1};

// 调用函数来调整 window_rect
AdjustWindowRectEx(&window_rect,
   GetWindowStyle(main_window_handle),
   GetMenu(main_window_handle) != NULL,
   GetWindowExStyle(main_window_handle));

// 将客户区偏移量赋给全局变量，以便在 DDraw_Flip()中使用
window_client_x0 = -window_rect.left;
window_client_y0 = -window_rect.top;

// 调用 MoveWindow()来调整窗口大小
MoveWindow(main_window_handle,
      CW_USEDEFAULT, // x 坐标
      CW_USEDEFAULT, // y 坐标
      window_rect.right - window_rect.left, // 宽度
      window_rect.bottom - window_rect.top, // 高度
      FALSE);

// 显示窗口
ShowWindow(main_window_handle, SW_SHOW);
} // end if windowed

// 执行游戏特有的所有初始化
Game_Init();

// 禁用 CTRL-ALT_DEL、 ALT_TAB
// 如果它导致系统崩溃，将其注释掉
SystemParametersInfo(SPI_SCREENSAVERRUNNING, TRUE, NULL, 0);

// 进入主事件循环
while(1)
  {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // 检查是否为退出消息
    if (msg.message == WM_QUIT)
      break;

    // 转换加速键
    TranslateMessage(&msg);

    // 将消息发送给 window proc
    DispatchMessage(&msg);
    } // end if

    // 主游戏处理工作
    Game_Main();

    } // end while

// 关闭游戏并释放所有的资源
Game_Shutdown();

// 启用 CTRL-ALT_DEL、ALT_TAB
// 如果这行代码导致系统崩溃，将其注释掉
SystemParametersInfo(SPI_SCREENSAVERRUNNING, FALSE, NULL, 0);

// 返回操作系统
return(msg.wParam);
```

```
} // end WinMain

// T3D II 游戏编程控制台函数////////////////

int Game_Init(void *parms)
{
// 在这里执行所有的游戏初始化工作

int index; // 循环变量

Open_Error_File("error.txt");

// 启动 DirectDraw，请根据需要替换其中的参数
DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP, WINDOWED_APP);

// 初始化 directinput
DInput_Init();

// 获取键盘控制权
DInput_Init_Keyboard();

// 初始化 directsound
DSound_Init();

// 加载声音
explosion_id = DSound_Load_WAV("exp1.wav");
laser_id     = DSound_Load_WAV("shocker.wav");

// 初始化 directmusic
DMusic_Init();

// 加载并启动主音轨
main_track_id = DMusic_Load_MIDI("midifile2.mid");
DMusic_Play(main_track_id);

// 在这里调用函数来获得对其他 directinput 设备的控制权

// 隐藏鼠标
ShowCursor(FALSE);

// 随机数生成器
srand(Start_Clock());

// 这里为初始化代码

// 创建系统颜色
rgb_green = RGB16Bit(0,255,0);
rgb_white = RGB16Bit(255,255,255);
rgb_blue  = RGB16Bit(0,0,255);
rgb_red   = RGB16Bit(255,0,0);

// 创建星空
for (index=0; index < NUM_STARS; index++)
  {
  // 将星星随机地放置在一个从视点(0,0,-d)到远裁剪面（0,0,far_z）的圆柱体内
  stars[index].x = -WINDOW_WIDTH/2  + rand()%WINDOW_WIDTH;
  stars[index].y = -WINDOW_HEIGHT/2 + rand()%WINDOW_HEIGHT;
  stars[index].z = NEAR_Z + rand()%(FAR_Z - NEAR_Z);

  // 设置星星的颜色
  stars[index].color = rgb_white;
  } // end for index

// 创建飞船模型

// 飞船的顶点列表
```

```
POINT3D temp_tie_vlist[NUM_TIE_VERTS] =
// 颜色和 x、y、z 坐标
{ {rgb_white,-40,40,0},    // p0
 {rgb_white,-40,0,0},    // p1
 {rgb_white,-40,-40,0},   // p2
 {rgb_white,-10,0,0},    // p3
 {rgb_white,0,20,0},     // p4
 {rgb_white,10,0,0},     // p5
 {rgb_white,0,-20,0},    // p6
 {rgb_white,40,40,0},     // p7
 {rgb_white,40,0,0},     // p8
 {rgb_white,40,-40,0}};   // p9

// 将模型复制到全局数组中
for (index=0; index<NUM_TIE_VERTS; index++)
   tie_vlist[index] = temp_tie_vlist[index];

// 飞船的边列表
LINE3D temp_tie_shape[NUM_TIE_EDGES] =
// 颜色、顶点 1、顶点 2
{ {rgb_green,0,2     },   // l0
 {rgb_green,1,3     },   // l1
 {rgb_green,3,4     },   // l2
 {rgb_green,4,5     },   // l3
 {rgb_green,5,6     },   // l4
 {rgb_green,6,3     },   // l5
 {rgb_green,5,8     },   // l6
 {rgb_green,7,9     } }; // l7

// 将模型复制到全局数组中
for (index=0; index<NUM_TIE_EDGES; index++)
   tie_shape[index] = temp_tie_shape[index];

// 初始化每艘飞船的位置和速度
for (index=0; index<NUM_TIES; index++)
  {
  // 初始化飞船
  Init_Tie(index);

  } // end for index

// 成功返回
return(1);

} // end Game_Init

//////////////////////////////////////////////////////////////////////

int Game_Shutdown(void *parms)
{
// 该函数关闭游戏，释放分配的所有资源

// 关闭

// 在这里释放为游戏分配的所有资源

// 关闭 directsound
DSound_Stop_All_Sounds();
DSound_Shutdown();

// 关闭 directmusic
DMusic_Delete_All_MIDI();
DMusic_Shutdown();

// 关闭 directinput
DInput_Shutdown();
```

```
// 最后关闭 directdraw
DDraw_Shutdown();

// 成功返回
return(1);
} // end Game_Shutdown

//////////////////////////////////////////////////////////////

void Start_Explosion(int tie)
{
// 根据传入的飞船模拟爆炸

// 找到第一个可用的爆炸结构
for (int index=0; index < NUM_EXPLOSIONS; index++)
  {
  if (explosions[index].state==0)
    {
    // 启用爆炸状态，并根据传入的飞船设置该爆炸结构

    explosions[index].state   = 1; // 启用爆炸状态
    explosions[index].counter = 0; // 重置爆炸计数器

    // 设置爆炸颜色
    explosions[index].color = rgb_green;

    // 复制边列表，以便模拟爆炸
    for (int edge=0; edge < NUM_TIE_EDGES; edge++)
      {
      // 边的起点
      explosions[index].p1[edge].x = ties[tie].x+tie_vlist
➡[tie_shape[edge].v1].x;
      explosions[index].p1[edge].y = ties[tie].y+tie_vlist
➡[tie_shape[edge].v1].y;
      explosions[index].p1[edge].z = ties[tie].z+tie_vlist
➡[tie_shape[edge].v1].z;

      // 边的终点
      explosions[index].p2[edge].x = ties[tie].x+tie_vlist
➡[tie_shape[edge].v2].x;
      explosions[index].p2[edge].y = ties[tie].y+tie_vlist
➡[tie_shape[edge].v2].y;
      explosions[index].p2[edge].z = ties[tie].z+tie_vlist
➡[tie_shape[edge].v2].z;

      // 计算边的弹道向量
      explosions[index].vel[edge].x = ties[tie].xv - 8+rand()%16;
      explosions[index].vel[edge].y = ties[tie].yv - 8+rand()%16;
      explosions[index].vel[edge].z = -3+rand()%4;

      } // end for edge

    return;
    } // end if found

  } // end for index

} // end Start_Explosion

//////////////////////////////////////////////////////////////

void Process_Explosions(void)
{
// 处理所有爆炸

// 遍历爆炸结构数组，渲染它们
for (int index=0; index<NUM_EXPLOSIONS; index++)
```

```
  {
  // 检查爆炸结构是否处于活动状态
  if (explosions[index].state==0)
    continue;

  for (int edge=0; edge<NUM_TIE_EDGES; edge++)
    {
    // 发生爆炸，更新边的起点坐标和终点坐标
    explosions[index].p1[edge].x+=explosions[index].vel[edge].x;
    explosions[index].p1[edge].y+=explosions[index].vel[edge].y;
    explosions[index].p1[edge].z+=explosions[index].vel[edge].z;

    explosions[index].p2[edge].x+=explosions[index].vel[edge].x;
    explosions[index].p2[edge].y+=explosions[index].vel[edge].y;
    explosions[index].p2[edge].z+=explosions[index].vel[edge].z;
    } // end for edge

  // 检查爆炸是否结束
  if (++explosions[index].counter > 100)
     explosions[index].state = explosions[index].counter = 0;

  } // end for index

} // end Process_Explosions

///////////////////////////////////////////////////////////

void Draw_Explosions(void)
{
// 绘制所有的爆炸结构

// 遍历所有的爆炸结构，并渲染它们
for (int index=0; index<NUM_EXPLOSIONS; index++)
  {
  // 检查爆炸结构是否处于活动状态
  if (explosions[index].state==0)
    continue;

  // 渲染爆炸结构
  // 每个爆炸结构都是由一系列边组成的
  for (int edge=0; edge < NUM_TIE_EDGES; edge++)
  {
  POINT3D p1_per, p2_per; // 用于存储透视投影点

  // 检查边是否在近裁剪面的外面
  if (explosions[index].p1[edge].z < NEAR_Z &&
    explosions[index].p2[edge].z < NEAR_Z)
    continue;

  // 第 1 步：对每个端点进行透视变换
  p1_per.x = VIEW_DISTANCE*explosions[index].p1[edge].x/explosions
➥[index].p1[edge].z;
  p1_per.y = VIEW_DISTANCE*explosions[index].p1[edge].y/explosions
➥[index].p1[edge].z;
  p2_per.x = VIEW_DISTANCE*explosions[index].p2[edge].x/explosions
➥[index].p2[edge].z;
  p2_per.y = VIEW_DISTANCE*explosions[index].p2[edge].y/explosions
➥[index].p2[edge].z;

  // 第 2 步：计算屏幕坐标
  int p1_screen_x = WINDOW_WIDTH/2  + p1_per.x;
  int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;
  int p2_screen_x = WINDOW_WIDTH/2 + p2_per.x;
  int p2_screen_y = WINDOW_HEIGHT/2 - p2_per.y;

  // 第 3 步：绘制边
  Draw_Clip_Line16(p1_screen_x, p1_screen_y, p2_screen_x, p2_screen_y,
```

```
            explosions[index].color,back_buffer, back_lpitch);

  } // end for edge

  } // end for index

} // end Draw_Explosions

//////////////////////////////////////////////////////////////

void Move_Starfield(void)
{
// 移动星空

int index; // 循环变量

// 从技术上说，星星应该是静止的
// 但我们将通过移动它们来模拟视点的移动
for (index=0; index<NUM_STARS; index++)
  {
  // 移动下一颗星星
  stars[index].z-=player_z_vel;

  // 检查是否在近裁剪面的外面
  if (stars[index].z <= NEAR_Z)
    stars[index].z = FAR_Z;

  } // end for index

} // end Move_Starfield

//////////////////////////////////////////////////////////////

void Draw_Starfield(void)
{
// 使用透视变换来绘制 3D 星星

int index; // 循环变量

for (index=0; index<NUM_STARS; index++)
  {
  // 绘制下一颗星星
  // 第 1 步：透视变换
  float x_per = VIEW_DISTANCE*stars[index].x/stars[index].z;
  float y_per = VIEW_DISTANCE*stars[index].y/stars[index].z;

  // 第 2 步：计算屏幕坐标
  int x_screen = WINDOW_WIDTH/2  + x_per;
  int y_screen = WINDOW_HEIGHT/2 - y_per;

  // 判断星星是否在屏幕外
  if (x_screen>=WINDOW_WIDTH || x_screen < 0 ||
    y_screen >= WINDOW_HEIGHT || y_screen < 0)
    {
    // 进入下一颗星星
    continue;
    } // end if
  else
    {
    // 渲染到缓存中
    ((USHORT *)back_buffer)[x_screen + y_screen*(back_lpitch >> 1)]
              = stars[index].color;
    } // end else

  } // end for index

} // Draw_Starfield
```

```
//////////////////////////////////////////////////////////

void Init_Tie(int index)
{
// 该函数将飞船放置到游戏世界的最远处

// 将飞船放在视景体内
ties[index].x = -WINDOW_WIDTH  + rand()%(2*WINDOW_WIDTH);
ties[index].y = -WINDOW_HEIGHT + rand()%(2*WINDOW_HEIGHT);
ties[index].z =  4*FAR_Z;

// 初始化飞船的速度
ties[index].xv = -4+rand()%8;
ties[index].yv = -4+rand()%8;
ties[index].zv = -4-rand()%64;

// 将飞船设置为活动状态
ties[index].state = 1;
} // end Init_Tie

//////////////////////////////////////////////////////////

void Process_Ties(void)
{
// 处理飞船并执行 AI
int index; // 循环变量

// 让每艘飞船向视点移动
for (index=0; index<NUM_TIES; index++)
  {
  // 飞船是否被击落
  if (ties[index].state==0)
    continue;

  // 进入下一艘飞船
  ties[index].z+=ties[index].zv;
  ties[index].x+=ties[index].xv;
  ties[index].y+=ties[index].yv;

  // 检查飞船是否穿过了近裁剪面
  if (ties[index].z <= NEAR_Z)
    {
    // 重新设置该飞船
    Init_Tie(index);

    // 将逃脱的飞船数加 1
    misses++;

    } // reset tie

  } // end for index

} // Process_Ties

//////////////////////////////////////////////////////////

void Draw_Ties(void)
{
// 使用透视变换以 3D 线框模式绘制飞船

int index; // 循环变量

// 用于计算飞船的包围框，以便进行碰撞检测
int bmin_x, bmin_y, bmax_x, bmax_y;

// 绘制每艘飞船
```

```
for (index=0; index < NUM_TIES; index++)
  {
  // 绘制下一艘飞船

  // 是否被击落?
  if (ties[index].state==0)
    continue;

  // 将包围框设置为不可能的值
  bmin_x =  100000;
  bmax_x = -100000;
  bmin_y =  100000;
  bmax_y = -100000;

  // 将飞船的 z 坐标除以最大的可能 z 值，得到一个 0～1 的值
  // 然后据此将 G 分量设置为 0～31 的值，使得飞船越近亮度越高
  USHORT rgb_tie_color = RGB16Bit(0,(31-31*(ties[index].z/(4*FAR_Z))),0);

  // 每艘飞船都由一系列边组成
  for (int edge=0; edge < NUM_TIE_EDGES; edge++)
  {
  POINT3D p1_per, p2_per; // 用于存储端点透视坐标

  // 第 1 步：对每个端点进行透视变换
  // 需要根据飞船相对于模型的位置相应地平移每个端点，这很重要
  p1_per.x =
      VIEW_DISTANCE*(ties[index].x+tie_vlist[tie_shape[edge].v1].x)/
      (tie_vlist[tie_shape[edge].v1].z+ties[index].z);

  p1_per.y = VIEW_DISTANCE*(ties[index].y+tie_vlist[tie_shape[edge].v1].y)/
        (tie_vlist[tie_shape[edge].v1].z+ties[index].z);

  p2_per.x = VIEW_DISTANCE*(ties[index].x+tie_vlist[tie_shape[edge].v2].x)/
        (tie_vlist[tie_shape[edge].v2].z+ties[index].z);

  p2_per.y = VIEW_DISTANCE*(ties[index].y+tie_vlist[tie_shape[edge].v2].y)/
        (tie_vlist[tie_shape[edge].v2].z+ties[index].z);

  // 第 2 步：计算屏幕坐标
  int p1_screen_x = WINDOW_WIDTH/2  + p1_per.x;
  int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;
  int p2_screen_x = WINDOW_WIDTH/2  + p2_per.x;
  int p2_screen_y = WINDOW_HEIGHT/2 - p2_per.y;

  // 第 3 步：绘制边
  Draw_Clip_Line16(p1_screen_x, p1_screen_y, p2_screen_x, p2_screen_y,
          rgb_tie_color,back_buffer, back_lpitch);

  // 根据这条边更新包围框
  int min_x = min(p1_screen_x, p2_screen_x);
  int max_x = max(p1_screen_x, p2_screen_x);

  int min_y = min(p1_screen_y, p2_screen_y);
  int max_y = max(p1_screen_y, p2_screen_y);

  bmin_x = min(bmin_x, min_x);
  bmin_y = min(bmin_y, min_y);

  bmax_x = max(bmax_x, max_x);
  bmax_y = max(bmax_y, max_y);

  } // end for edge

  // 检查飞船是否被激光击中
  if (cannon_state==1)
    {
    // 检查激光瞄准位置是否在包围框内
```

```
    if (target_x_screen > bmin_x && target_x_screen < bmax_x &&
      target_y_screen > bmin_y && target_y_screen < bmax_y)
     {
     // 飞船被击中
     Start_Explosion(index);

     // 播放爆炸声
     DSound_Play(explosion_id );

     // 增加得分
     score+=ties[index].z;

     // 将被击落的飞船数加 1
     hits++;

     // 重新初始化该飞船
     Init_Tie(index);

     } // end if

    } // end if

  } // end for index

} // end Draw_Ties

///////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
// 这是游戏的核心，将不断地被实时调用
// 就像 C 语言中的 main()函数，游戏中所有的函数调用都是在这里进行的

int       index;      // 循环变量

// 启动定时时钟
Start_Clock();

// 清空绘制表面(drawing surface)
DDraw_Fill_Surface(lpddsback, 0);

// 读取键盘和其他设备输入
DInput_Read_Keyboard();

// 游戏逻辑

if (game_state==GAME_RUNNING)
{
// 移动准星
if (keyboard_state[DIK_RIGHT])
  {
  // 将准星向右移动
  cross_x+=CROSS_VEL;

  // 检查是否超出了窗口
  if (cross_x > WINDOW_WIDTH/2)
   cross_x = -WINDOW_WIDTH/2;

  } // end if
if (keyboard_state[DIK_LEFT])
  {
  // 将准星向左移
  cross_x-=CROSS_VEL;

  // 检查是否超出了窗口
  if (cross_x < -WINDOW_WIDTH/2)
   cross_x = WINDOW_WIDTH/2;
```

```
  } // end if
if (keyboard_state[DIK_DOWN])
  {
  // 将准星向上移
  cross_y-=CROSS_VEL;

  // 检查是否超出了窗口
  if (cross_y < -WINDOW_HEIGHT/2)
   cross_y = WINDOW_HEIGHT/2;
  } // end if
if (keyboard_state[DIK_UP])
  {
  // 将准星向下移
  cross_y+=CROSS_VEL;

  // 检查是否超出了窗口
  if (cross_y > WINDOW_HEIGHT/2)
   cross_y = -WINDOW_HEIGHT/2;
  } // end if

// 控制飞船的速度
if (keyboard_state[DIK_A])
  player_z_vel++;
else
if (keyboard_state[DIK_S])
  player_z_vel--;

// 检测玩家是否开火
if (keyboard_state[DIK_SPACE] && cannon_state==0)
  {
  // 开火
  cannon_state = 1;
  cannon_count = 0;

  // 存储上一次的瞄准位置
  target_x_screen = cross_x_screen;
  target_y_screen = cross_y_screen;

  // 播放激光炮开火的声音
  DSound_Play(laser_id);

  } // end if

} // end if game running

// 处理激光炮，这是一个游戏状态机，在就绪、开火、冷却状态之间切换

// 开火阶段
if (cannon_state == 1)
  if (++cannon_count > 15)
   cannon_state = 2;

// 冷却阶段
if (cannon_state == 2)
  if (++cannon_count > 20)
   cannon_state = 0;

// 移动星星
Move_Starfield();

// 移动飞船并执行飞船 AI
Process_Ties();

// 处理爆炸
Process_Explosions();

// 锁定后缓存，并获得指针和宽度
```

```
DDraw_Lock_Back_Surface();

// 绘制星空
Draw_Starfield();

// 绘制飞船
Draw_Ties();

// 模拟爆炸
Draw_Explosions();

// 绘制准星

// 首先计算准星的屏幕坐标
// 注意 y 轴倒转过来了
cross_x_screen = WINDOW_WIDTH/2  + cross_x;
cross_y_screen = WINDOW_HEIGHT/2 - cross_y;

// 使用屏幕坐标绘制准星
Draw_Clip_Line16(cross_x_screen-16,cross_y_screen,
         cross_x_screen+16,cross_y_screen,
         rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen,cross_y_screen-16,
         cross_x_screen,cross_y_screen+16,
         rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen-16,cross_y_screen-4,
         cross_x_screen-16,cross_y_screen+4,
         rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen+16,cross_y_screen-4,
         cross_x_screen+16,cross_y_screen+4,
         rgb_red,back_buffer,back_lpitch);

// 绘制激光束
if (cannon_state == 1)
  {
  if ((rand()%2 == 1))
   {
   // 右激光束
   Draw_Clip_Line16(WINDOW_WIDTH-1, WINDOW_HEIGHT-1,
        -4+rand()%8+target_x_screen,-4+rand()%8+target_y_screen,
        RGB16Bit(0,0,rand()),back_buffer,back_lpitch);
   } // end if
  else
   {
   // 左激光束
   Draw_Clip_Line16(0, WINDOW_HEIGHT-1,
        -4+rand()%8+target_x_screen,-4+rand()%8+target_y_screen,
        RGB16Bit(0,0,rand()),back_buffer,back_lpitch);
   } // end if

  } // end if

// 渲染结束，解除对后缓存的锁定
DDraw_Unlock_Back_Surface();

// 绘制信息
sprintf(buffer, "Score %d    Kills %d     Escaped %d", score, hits, misses);
Draw_Text_GDI(buffer, 0,0,RGB(0,255,0), lpddsback);

if (game_state==GAME_OVER)
  Draw_Text_GDI("G A M E  O V E R", 320-8*10,240,RGB(255,255,255),lpddsback);

// 检查音乐是否播放完毕，如果是重新播放
if (DMusic_Status_MIDI(main_track_id)==MIDI_STOPPED)
```

```
  DMusic_Play(main_track_id);

// 交换前后缓存
DDraw_Flip();

// 同步到 30 帧/秒
Wait_Clock(30);

// 判断游戏是否结束
if (misses > 100)
  game_state = GAME_OVER;

// 检查用户是否要退出
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // end if
// 成功返回
return(1);
} // end Game_Main

//////////////////////////////////////////////////////////////////////
```

对于 3D 游戏来说，这些代码是相当精简的！这是一个完整的 3D Win32/DirectX 游戏。尽管在 T3DLIB 模块中有数千行代码，但我们不需要编写它，DirectX 和其他人完成了这些代码。

开始分析代码之前，希望读者自己动手编译该程序。成功编译该程序之前，请不要阅读后面的内容！您只需遵循前面概述的步骤，设置编译器，使之创建 Win32.EXE 应用程序，为 DirectX 设置适当的搜索路径和链接列表。然后，一旦准备好工程，必须包括下面这些源文件：

- T3DLIB1.CPP；
- T3DLIB2.CPP；
- T3DLIB3.CPP；
- RAIDERS3D.CPP。

当然，还需要将下述头文件放在编译器的工作目录下：

- T3DLIB1.H；
- T3DLIB2.H；
- T3DLIB3.H。

最后，还必须将 DirectX .LIB 文件包括在工程（链接列表）中。您只需要下列 DirectX .LIB 文件：

- DDRAW.LIB；
- DSOUND.LIB；
- DINPUT.LIB；
- DINPUT8.LIB。

可以将生成的.EXE 文件命名为任何名称，如 TEST.EXE 或 RAIDERS3D_TEST.EXE，但在成功编译之前，请不要阅读后面的内容。

1.5.1　事件循环

WinMain()是所有 Windows 程序的主入口，就像 main()是所有 DOS/Unix 程序的入口一样。Raiders3D 的 WinMain()创建一个窗口，然后进入事件循环。如果 Windows 需要做某些事情，它将会执行。WinMain()首先创建一个 Windows 类，然后注册这个类。接下来创建游戏窗口，然后调用函数 Game_Init()，处理游戏的所有初始化操作。完成所有初始化操作后，进入标准的 Windows 事件循环以检测消息。收到消息后，调用 Windows 例程 WinProc 处理该消息；否则，调用主游戏编程函数 Game_Main()，其中将执行真正的游戏操作。

注意：阅读过《Windows 游戏编程大师技巧》的读者将注意到，WinMain( )的初始化代码中有一些额

外的代码，用于处理窗口图形和调整窗口大小。这种新功能和 16 位颜色模式支持，是新版本游戏引擎 T3DLIB 的一部分。但是，本书的大多数 3D 将支持 8 位颜色图形，因为 16 位颜色软件 3D 的速度仍然不能令人满意。

如果愿意，可以永久性地在 Game_Main() 中循环，而不返回到 WinMain() 中的主事件循环。这不是一种好的做法，因为这样 Windows 将永远不会收到任何消息。我们要做的是执行一个动画帧或逻辑，然后返回到 WinMain()。这样，Windows 将继续负责处理消息。该过程如图 1.17 所示。

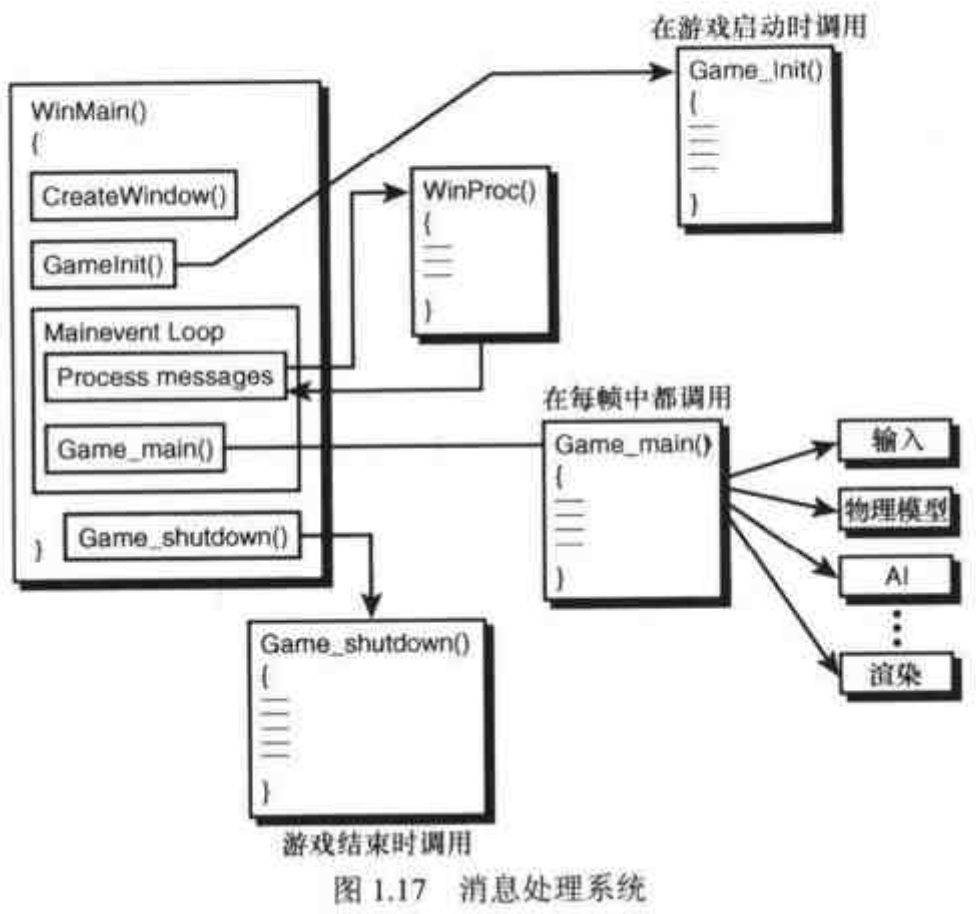


图 1.17　消息处理系统

1.5.2　核心 3D 游戏逻辑

进入 Game_Main() 后，将执行 *Raiders3D* 的逻辑，在后缓存中渲染 3D 游戏图像。在循环的最后，通过调用 DDraw_Flip()，将生成的图像显示在显示器上，产生动画影像。游戏循环由“2D/3D 游戏的元素”一节中介绍的标准部分组成。这里重点介绍 3D 图形学。

敌方 AI 的游戏逻辑相当简单，在 3D 空间中视图外面的某个随机位置生成一艘敌方飞船。图 1.18 显示了 *Raiders3D* 的 3D 空间，如图中所示，相机（视点）固定在(0，0，-zd)处，其中 zd 是视距，它是视点离图像被投影到其上的虚拟窗口的距离。这个 3D 空间使用的是左手坐标系统，因为正 *z* 轴指向屏幕里面。

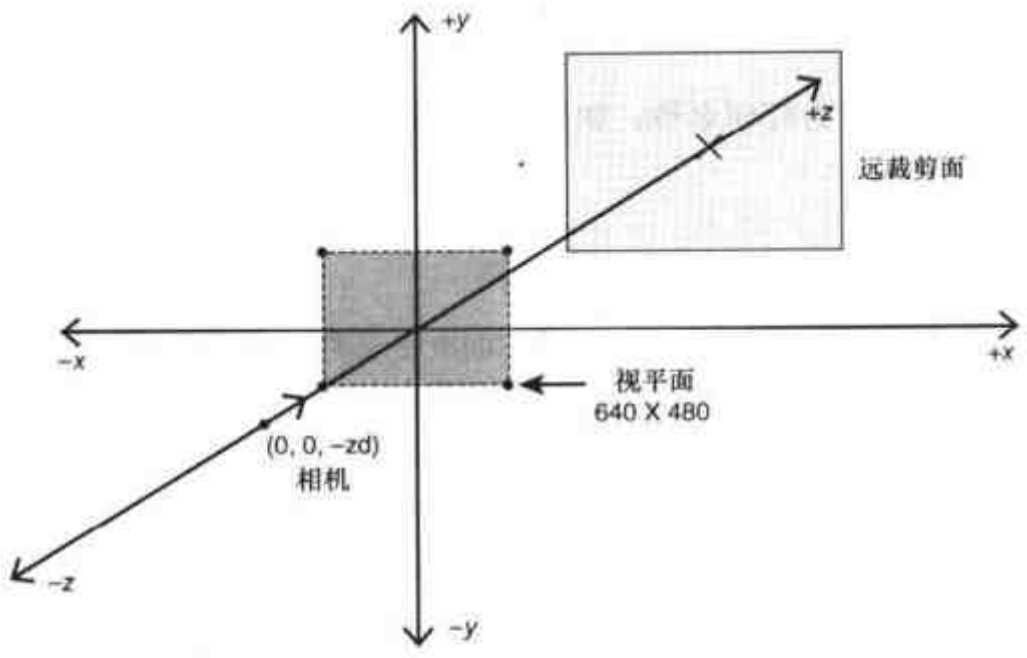
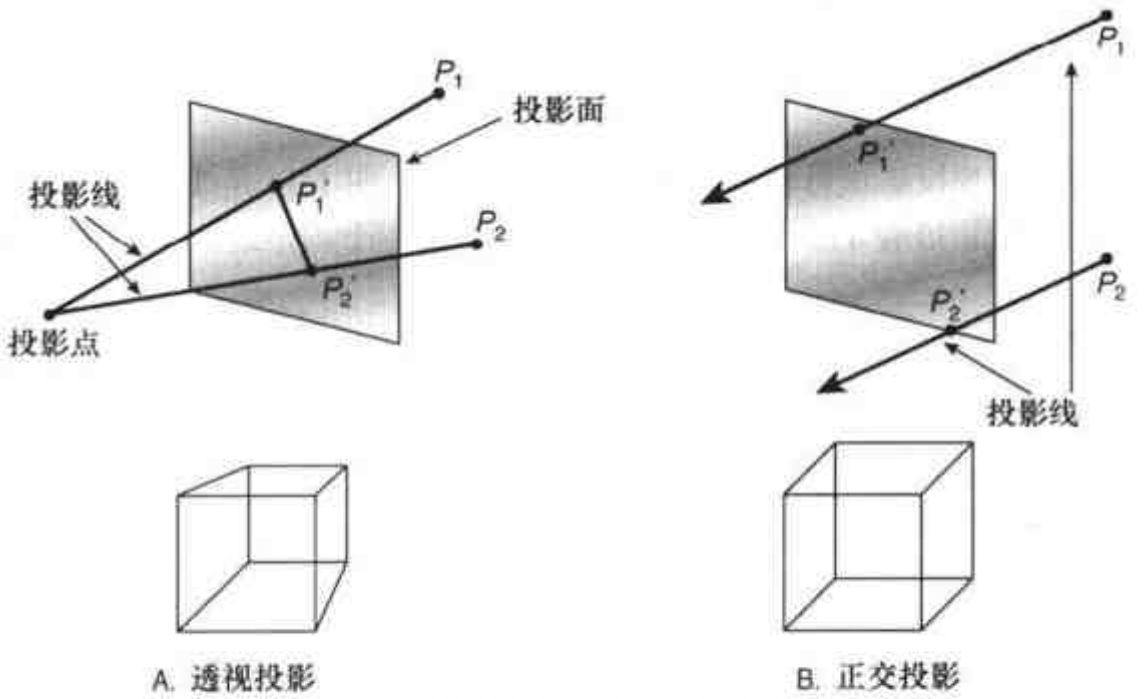


图 1.18　*Raiders3D* 的游戏世界

在敌方飞船生成之后，它将沿一条固定路径运动，该路径与玩家的观察位置相交，因而是一个碰撞过程。路径和初始位置在函数 Init_Tie() 中生成。玩家的目标是对准敌方飞船并开火。

问题是 3D 图像是如何生成的呢？敌方飞船是多边形物体，它定义了 3D 物体的外形。也就是说，它们是 2D 轮廓，而不是完全的 3D 物体。3D 的关键是透视投影。图 1.19 说明了正交投影与透视投影的差别，它们是 3D 图形系统中的两种基本投影类型。



A. 透视投影　　B. 正交投影

图 1.19　正交投影与透视投影

1.5.3　3D 投影

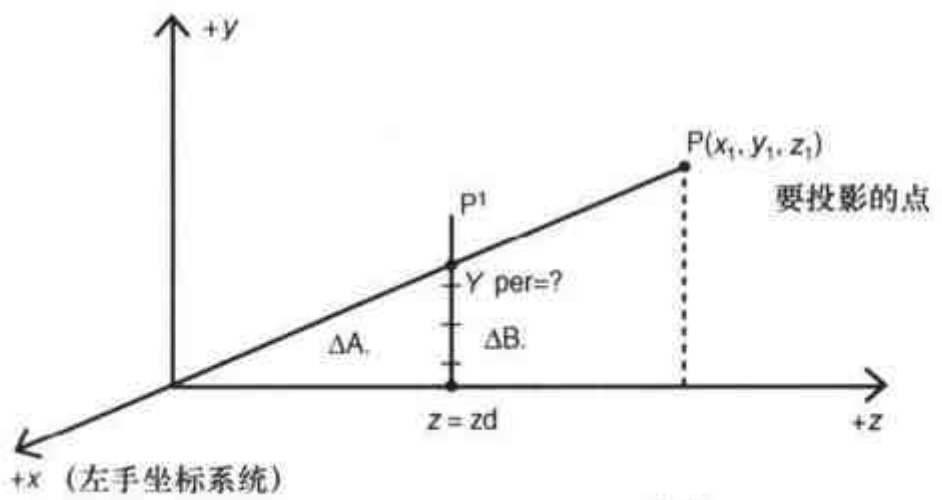
对于技术图样和图像，正交投影特别适合，而透视投影则不太适合。正交投影的数学原理相当简单，基本上是将每个点 z 坐标丢弃，如下所示：

给定 3D 空间中任一点 (x, y, z)：

$$x_{ortho} = x$$

$$y_{ortho} = y$$

透视投影有点复杂，这里不打算详细介绍其原理。一般来说，计算 2D 透视投影坐标时需要考虑 z 坐标和视距，其数学原理如图 1.20 所示，它是基于三角形相似的。如图所示，点 (x_{per}, y_{per}) 可以用下列方程计算。



ΔA 与 ΔB 相似，

因此：

$$\frac{Y\ per}{Y_1} = \frac{zd}{z}$$

$$Y\ per = Y_1 \cdot \frac{zd}{z}$$

图 1.20　投影变换

给定 3D 空间中的(*x*，*y*，*z*)和视距 zd：

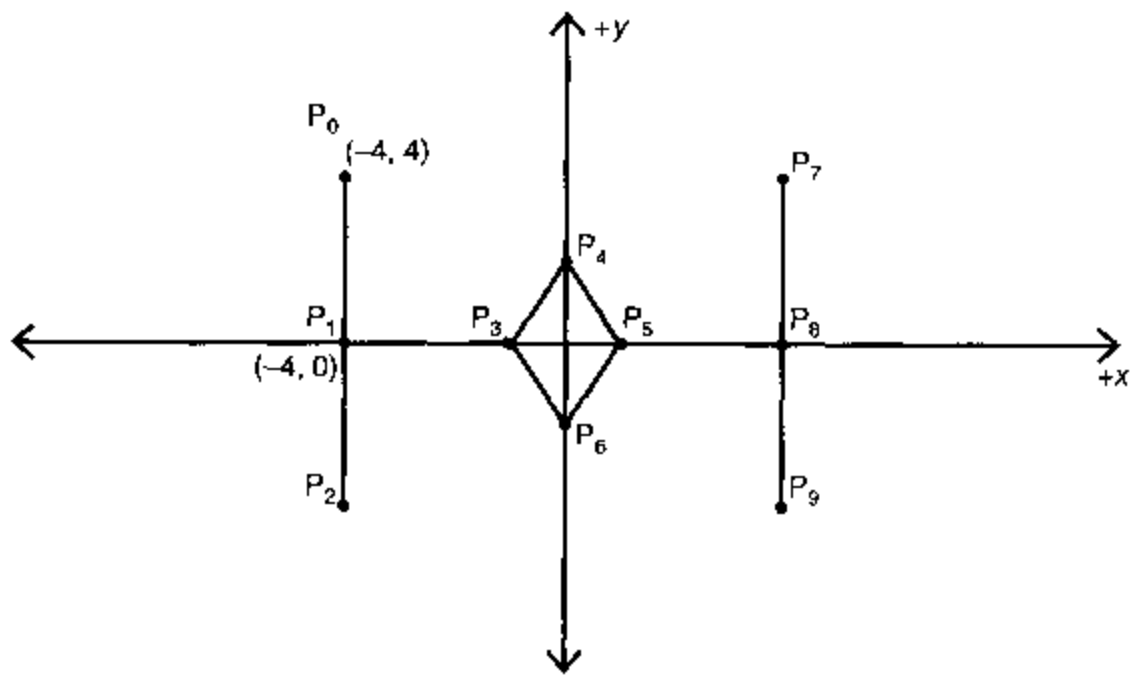
$$x_{per} = zd*x/z$$

$$y_{per} = zd*y/z$$

可以像移动 2D 物体那样移动 3D 多边形物体，在渲染之前使用透视投影方程对点进行变换，让物体能够在 3D 空间正确地显示和移动。当然，将坐标(x_{per}，y_{per})转换为实际屏幕坐标将涉及许多细节，但就现在而言，这些并不重要。您只需要知道 3D 物体以 3D 方式移动，并使用数学方法（考虑了透视）将 3D 物体投影到 2D 观察面（屏幕）上。

每艘敌方飞船将使用透视投影进行转换，并渲染到屏幕上。图 1.21 是 TIE 飞船的线框模型，基本上，渲染是基于这个模型完成的。每艘 TIE 飞船的实际位置存储在一个由下述结构组成的数组中：

```
//TIE 飞船
typedef struct TIE_TYP
    {
    int state;   // 飞船的状态，0 表示死亡，1 表示活着
    float x, y, z; // 飞船的位置
    float xv,yv,zv; // 飞船的速度
    } TIE, *TIE_PTR;
```



顶点列表		边列表	
P_0	(–4, 4)	边 0	P_0–P_2
P_1	(–4, 0)	边 1	P_1–P_3
P_2	(–4, –4)	边 2	P_3–P_4
P_3	(–1, 0)	边 3	P_4–P_5
P_4	(0, 2)	边 4	P_5–P_6
P_5	(1, 0)	边 5	P_6–P_3
P_6	(0, –2)	边 6	P_5–P_8
P_7	(4, 4)	边 7	P_7–P_9
P_8	(4, 0)		
P_9	(4, –4)		

图 1.21　TIE 飞船的线框模型

需要绘制敌方飞船时，使用 TIE 结构中的数据、TIE 3D 模型和透视投影。

TIE 飞船越近，亮度越高。这种效果很容易实现。实际上，程序使用 *z* 距离作为缩放因子来控制 TIE 飞船的亮度。

1.5.4　星空

星空实际上就是由空间外部一个发射源产生的点集。它们穿过玩家的视点后将被回收。这些点通过透视投影，被渲染为完全的 3D 物体，但大小为 1×1×1 像素，它们是真正的点，总是为单个像素。

1.5.5　激光炮和碰撞检测

玩家用以击落敌方飞船的激光炮实际上是从屏幕角落投影 2D 的直线，并会聚到十字准星。通过观察屏幕上所有 3D 飞船的 2D 投影，并测试激光是否在每个投影的包围框中，来完成碰撞检测，如图 1.22 所示。

由于激光是以光速前进的，所以该算法是可行的。不管目标是在 10m 以外还是在 10000km 以外，只要用激光束瞄准它，使得激光束与 3D 图像的投影相交，就会发生碰撞。

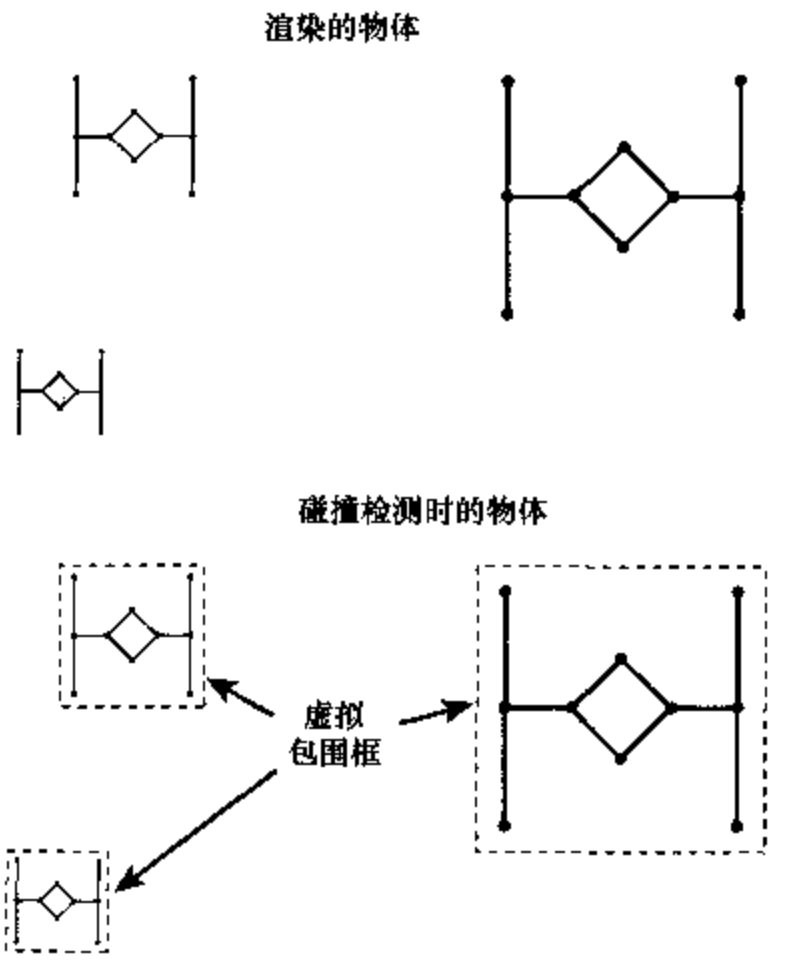


图 1.22　使用 2D 投影包围框进行碰撞检测

1.5.6　爆炸

相对于代码量而言，这个游戏的爆炸效果很不错。飞船被激光脉冲击中时，将组成飞船 3D 模型的直线复制另一个数据结构中，然后随机移动这些直线，仿佛它们是飞船的碎片。这些碎片将移动几秒钟，然后爆炸结束。爆炸效果相当真实，用于实现这种效果的代码不超过 100 行。

1.5.7　玩 *Raiders3D*

只要双击附带光盘中的 RAIDERS3D.EXE，该游戏便会立即启动。控制方法如下。

- 方向键：移动准星。
- 空格键：开火。
- Esc：退出游戏。

注意：逃脱的敌方飞船超过 100 艘时，游戏将结束！

该游戏使用了 DirectDraw、DirectInput、DirectSound 和 DirectMusic，因此需要确保在系统上正确安装了 DirectX 运行库。

注意：本书后面将使用支持光照效果、3D 模型和纹理映射的多边形引擎，重新编写 Raiders3D，让读者能够比较两者之间的差别！

1.6　总　　结

本章简要地介绍了 3D 游戏编程，其中最重要的内容是如何编译 DirectX 程序。但是，本章确实介绍了有关游戏编程的知识、游戏循环、通用 3D 透视变换以及如何使用编译器。另外还介绍了《Windows 游戏编程大师技巧》中的 T3DLIB 游戏引擎，我们使用它来帮助设置 DirectX 系统，使之正确运行，从而可以将重点放在 3D 游戏编程上，而无需考虑创建表面、查询接口和加载声音等细节。能够正确编译该程序后，请尝试添加更多的敌方飞船、星星等。

第 2 章　Windows 和 DirectX 简明教程

本章简要地介绍 Windows 和 DirectX 编程。本书假定您熟悉 Win32 编程和 DirectX 编程，但如果您不了解有关 Win32/DirectX 编程的知识，也可以使用我创建的 API，并将重点放在 3D 内容上。本章为那些不熟悉 Win32/DirectX 编程的读者提供这方面的简明课程，将介绍以下内容：

- Windows 编程；
- 事件循环；
- 编写一个简单的 Windows 程序；
- DirectX 基础；
- 组件对象模型。

2.1　Win32 编程模型

Windows 是一个多任务多线程操作系统，也是一个事件驱动型操作系统。与大多数 DOS 程序不同，大多数 Windows 程序都等待用户做一些事情（从而引发事件），然后 Windows 响应该事件并采取行动，如图 2.1 所示。图 2.1 显示了很多应用程序窗口，它们分别发送事件或消息给 Windows 进行处理。Windows 处理一些消息或事件，但大多数消息或事件都传回应用程序进行处理。

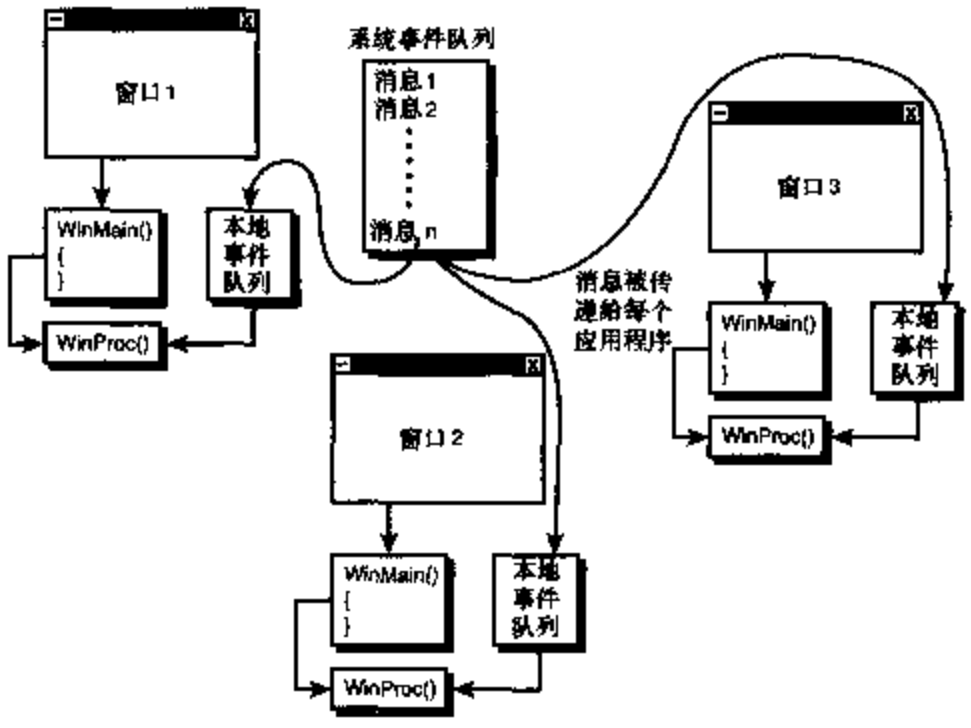


图 2.1　Windows 事件处理

大多数情况下，您不需关心正在运行的其他应用程序，Windows 将处理它们。您只需关心自己的应用程序，并处理其窗口中的消息或事件。在 Windows 3.0/3.1 中，并不完全如此，这些版本的 Windows 并非

真正的多任务操作系统，执行一个应用程序时不能执行其他的应用程序。在这些版本下运行的应用程序有些迟滞。如果其他应用程序过分占用了系统，您的应用程序将不能做任何事情。但是，Windows 9x/Me/2000/XP/NT 没有这样的问题，这些操作系统会在它任何合适的时候改变应用程序原有的计划——当然，由于改变速度非常快，用户根本觉察不到。

注意：至此，读者了解了所需的所有操作系统概念。幸运的是，Windows 是当今比较适合于编写游戏的操作系统，您无需关心调度等问题，而只需编写游戏代码并将机器的性能发挥到极限。

2.2　Windows 程序的最小需求

现在您已经初步了解了 Windows 操作系统及其一些属性和底层设计，下面介绍我们的第一个 Windows 程序。

在任何一种新语言或操作系统中编写一个“Hello World”程序已成为惯例，我们将遵循这种惯例。下面的代码清单是一个标准的基于 DOS 的“Hello World”程序。

```
// DEMOII2_1.CPP: "hello world"程序的标准版本
#include <stdio.h>

// 所有标准 DOS/控制台程序的入口
void main(void)
{
printf("\nTHERE CAN BE ONLY ONE!!!\n");
} // end main
```

下面来看一看如何编写这样的 Windows 程序。

注意：要编译 DEMOII2_1.CPP，可以使用 Visual C++或 Borland 编译器创建一个“控制台应用程序”(console application)。它们与 DOS 应用程序类似，但是 32 位程序。它们只在文本模式下运行，非常适合用于测试算法和想法。

一切从 WinMain( )开始

所有 Windows 程序都从执行一个名为 WinMain()的函数开始，它类似于 DOS/Unix 程序中的 main()函数。在 WinMain()中做哪些事情完全取决于您，如果愿意，也可以创建一个窗口、开始处理事件、在屏蔽上绘制图像等。另一方面，也可以只调用 Win32 API 函数，这正是我们要做的。

例如，我只想通过消息框在屏幕上显示一些内容。正好有一个 Win32 API 函数可以完成这项任务：MessageBox()。下面的代码清单是一个完整的可编译的 Windows 程序，它创建并显示一个消息框，用户可以移动或关闭该消息框。

```
// DEMOII2_2.CPP--一个简单的消息框
#define WIN32_LEAN_AND_MEAN // 不使用 MFC

#include <windows.h>    // Windows 头文件
#include <windowsx.h>    // 包含大量的宏
// 所有 Windows 程序的入口
int WINAPI WinMain(HINSTANCE hinstance,
          HINSTANCE hprevinstance,
       LPSTR lpcmdline,
       int ncmdshow)
{
// 调用显示消息框的 API，并将父窗口句柄设置为 NULL
MessageBox(NULL, "THERE CAN BE ONLY ONE!!!",
          "MY FIRST WINDOWS PROGRAM",
          MB_OK | MB_ICONEXCLAMATION);

// 退出程序
```

```
return(0);

} // end WinMain
```

编译该程序的步骤如下：

1．新建一个 Win32.EXE Application Project，并包括附带光盘中 T3DIICHAP02 目录下的 DEMOII2_2.CPP 文件；

2．编译并链接该程序；

3．运行该程序（也可以运行附带光盘中已编译好的 DEEMOII2_2.EXE）。



图 2.2　DEMOII2_2.EXE 运行时的屏幕截图

编译并运行该程序后，将看到如图 2.2 所示的画面。

有了完整的 Windows 程序后，下面将逐行分析该程序。

第 1 行代码如下：

```
#define WIN32_LEAN_AND_MEAN
```

有必要解释一下这行代码。可以使用两种方法来编写 Windows 程序：使用 MFC（Microsoft Foundation Classes）或 SDK（Software Development Kit）。MFC 要复杂得多，它完全基于 C++和类，就编写游戏而言，完全没必要使用 MFC。另一方面，SDK 更易于管理，可以在一两周内学会，且使用 C 语言。因此，本书将使用 SDK。

回到前面那行代码：WIN32_LEAN_AND_MEAN 指示编译器不要包含无关的 MFC 文件。

接下来的两行代码用于包含头文件：

```
#include <windows.h>
#include <windowsx.h>
```

第 1 行包含 windows.h，它实际上将包含所有的 Windows 头文件——有很多头文件，因此它更像是一个嵌套宏，可避免您不必手工包含多个头文件。

第 2 行包含 windowsx.h，它是一个宏和常量头文件，其中包含了大量的宏和常量，让 Windows 编程更加方便。

接下来是程序最重要的组成部分：Windows 应用程序的主入口——WinMain()：

```
int WINAPI WinMain(HINSTANCE hinstance,
        HINSTANCE hprevinstance,
      LPSTR lpcmdline,
      int ncmdshow);
```

首先，您将注意到奇怪的声明符号 WINAPI，它等价于函数声明符号 PASCAL，强制按从左至右的次序处理参数，而不像默认的 CDECL 那样按从右至左的次序处理参数。但是，调用规则声明符号 PASCAL 已经废弃，而使用 WINAPI。对于 WinMain()函数，必须使用 WINAPI；否则，启动代码将不能正确地将参数传递给它！

接下来详细介绍其中的每个参数：

- hinstance——该参数是 Windows 为应用程序生成的实例名柄。实例（Instance）是用于跟踪资源的指针（或数字）。在这里，hinstance 用于跟踪应用程序，就像姓名或地址。应用程序执行时，Windows 将该参数提供给应用程序。
- hprevinstance——该参数已不再使用，但在以前的 Windows 版本中，它用于跟踪应用程序的前一个实例，即启动当前应用程序的应用程序实例。不要奇怪为什么 Microsoft 要停止使用该参数，它确实令人很头痛！
- lpcmdline——一个以 NULL 结尾的字符串，类似于标准 C/C++函数 main(int argc，char ** argv)中的命令行参数，但没有单独的 argc 用于指示命令行参数数量。例如，如果您创建了一个 Windows 应用程序，将其命名为 TEST.EXE，并以如下命令行参数启动该程序：

```
TEST.EXE one two three
```

lpcmdline 将包含下列数据：

```
lpcmdline = "one two three"
```

注意，.EXE 文件的名称并不会包含在命令行参数中。

● ncmdshow——最后一个参数是启动期间传递给应用程序的一个整数，指出如何打开主应用程序窗口。因此，用户无法控制应用程序如何启动。当然，作为程序员，可以不使用该参数。表 2.1 列出了参数 ncmdshow 的几种常见值。

表 2.1　　ncmdshow 的 Windows 编码

值	含义
SW_SHOWNORMAL	激活并显示一个窗口。如果该窗口被最小化或最大化，Windows 将它恢复到原来的大小和位置。应用程序在首次显示窗口时应该使用该标记
SW_SHOW	激活窗口并按当前大小和位置显示它
SW_HIDE	隐藏窗口并激活另一个窗口
SW_MAXIMIZE	最大化指定的窗口
SW_MINIMIZE	最小化指定的窗口
SW_RESTORE	激活并显示一个窗口。如果该窗口被最小化或最大化，Windows 将它恢复到原来的大小和位置。应用程序在恢复一个最小化窗口时应该使用该标记
SW_SHOWMAXIMIZED	激活一个窗口并以最大化方式显示它
SW_SHOWMINIMIZED	激活一个窗口并以最小化方式显示它
SW_SHOWMINNOACTIVE	以最小化方式显示窗口，原来活动的窗口仍保持活动状态
SW_SHOWNA	以当前状态显示一个窗口，原来活动的窗口仍保持活动状态
SW_SHOWNOACTIVATE	以上一次的大小和位置显示一个窗口，原来活动的窗口仍保持活动状态

如表 2.1 所示，对于 ncmdshow 参数有很多种设置（其中大多数在目前都没有意义）。实际上，大多数从来不会用于设置 ncmdshow 参数；您将在另一个函数 ShowWindows() 中使用它们，该函数在窗口被创建后实际显示它，这将在本章后面介绍。

需要指出的是，Windows 有很多您从来不会使用的选项、标记等，但它们仍然存在。这就像 VCR 编程选项，选项越多越好，如果您不想使用，不使用就是了。Windows 就是以这种方式设计的，它必须满足每个人的需要，因此需要提供很多选项。实际上，我们在绝大多数时候只会用到 SW_SHOW、SW_SHOWNORMAL 和 SW_HIDE。

最后，来看一下 WinMain() 函数中的函数调用 MessageBox()。它为我们完成了全部工作。MessageBox() 是一个可以为我们做很多事情的 Win32 API 函数，这样我们就不必自己做这些事情。MessageBox() 函数用于以不同的图标和按钮来显示一些消息。正如您知道的，在 Windows 应用程序中显示简单消息非常普遍，因此这个函数可以为程序员节省大量的时间。

MessageBox() 的功能不多，但确实能够在屏幕上显示一个窗口、询问问题并等待用户的输入。下面是 MessageBox() 的原型：

```
int MessageBox( HWND  hwnd,   // 父窗口句柄
        LPCTSTR lptext;  // 要显示在消息框中的文本
        LPCTSTR lpcaption,// 消息框的标题
        UINT  utype);  // 消息框的样式
```

● hwnd——包含消息框的窗口的句柄。到现在为止，本书还没有介绍窗口句柄，所以您只需将它理解为消息框的父窗口。在程序 DEMOII2_2.CPP 中，我们将它设置为 NULL，表示使用 Windows 桌面作为其父窗口。

● lptext——是一个以 NULL 结尾的字符串，包含要显示的文本。

● lpcaption——是一个以 NULL 结尾的字符串，包含消息框的标题。

● utype——该参数指定显示哪种类型的消息框。表 2.2 列出了多种类型的消息框（不完全）。

表 2.2 MessageBox( )选项

标记	含义
MB_OK	只包含一个按钮：OK，这是默认值
MB_OKCANCEL	包含两个按钮：OK 和 Cancel
MB_RETRYCANCEL	包含两个按钮：Retry 和 Cancel
MB_YESNO	包含两个按钮：Yes 和 No
MB_YESNOCANCEL	包含三个按钮：Yes、No 和 Cancel
MB_ABORTRETRYIGNORE	包含三个按钮：Abort、Retry 和 Ignore
MB_ICONEXCLAMATION	在消息框上显示一个惊叹号图标
MB_ICONINFORMATION	在消息框上显示一个消息图标——在一个圆圈中显示小写字母 i
MB_ICONQUESTION	在消息框上显示一个问号图标
MB_ICONSTOP	在消息框上显示一个停止符号图标
MB_DEFBUTTONn	*n* 为 1～4 的数字，表示默认情况下第 *n* 个按钮处于活动状态（按从左至右的次序编号）

还有一些其他的操作系统级标记，但我们不关心它们，详情可参阅编译器的 Win32 SDK 联机帮助。

对于表 2.2 中的值，可以使用逻辑 OR 将它们组合起来，以设置所需类型的消息框。通常情况下，只需要从每组选择该值，然后使用逻辑 OR 将它们组合起来。

像所有良好的 Win32 API 函数一样，MessageBox() 也返回一个值，让您了解消息框中发生的情况。对于 Yes/No 询问型消息框，您可能想知道返回值。可能返回的值如表 2.3 所示。

表 2.3 MessageBox( )的返回值

值	含义	值	含义
IDABORT	用户选择了 Abort 按钮	IDOK	用户选择了 OK 按钮
IDCANCEL	用户选择了 Cancel 按钮	IDRETRY	用户选择了 Retry 按钮
IDIGNORE	用户选择了 Ignore 按钮	IDYES	用户选择了 Yes 按钮
IDNO	用户选择了 No 按钮		

完成对第一个 Windows 程序代码的逐行分析后，您可以试着修改该程序，并以不同的方式编译它：尝试修改各种编译器选项，如优化选项，代码生成选项等。然后尝试通过调试器运行该程序，看能否了解程序的执行过程。

要听到声音效果，一种很简单的方法是使用 MessageBeep() 函数。可以在 Win32 SDK 中查找该函数的相关信息，其用法与 MessageBox() 函数一样简单。下面是这个函数：

```
BOOL MessageBeep(UINT utype); // 要播放的声音
```

可以使用如表 2.4 所示的常量来设置声音。

表 2.4 MessageBeep( )函数的声音标识符

值	含义	值	含义
MB_ICONASTERISK	系统 Asterisk 声音	MB_ICONQUESTION	系统 Question 声音
MB_ICONEXCLAMATION	系统 Exclamation 声音	MB_OK	系统默认声音
MB_ICONHAND	系统 Hand 声音	0xFFFFFFFF	使用计算机扬声器的标准蜂鸣声

如果安装了 MS-Plus 主题方案，声音将更有趣。

现在我们了解了 Win32 API 的功能，有数百个 Win32 API 函数。当然，它们并不是运行速度最快的函

数，但对于常规清理工作、I/O 和 GUI，Win32 API 是非常方便的。

现在总结一下至此介绍过的 Windows 编程知识。首先，Windows 是多任务多线程操作系统，可同时运行多个应用程序。然而，我们无需负责处理所有的事情。与我们密切相关的是，Windows 是事件驱动的。这意味着需要处理事件（到现在为止我们并不知道如何处理事件），并对事件做出响应。最后，所有 Windows 程序都从函数 WinMain() 开始执行，该函数的参数比 DOS main() 多几个，但在合理的范围内。

下面编写一个非常基本的 Windows 应用程序，实际上本章后面编写游戏引擎时将以此为基础。

2.3　一个基本的 Windows 应用程序

本书的目标是编写在 Windows 平台上运行的 3D 游戏，所以无需知道很多有关 Windows 编程的知识。实际上，需要的只是一个框架 Windows 程序，它打开窗口、处理消息并调用主游戏循环，仅此而已。本节的首要目标是介绍如何创建简单的 Windows 程序，但同时为如何编写类似于 32 位 DOS/Unix 机器的游戏编程“Shell”做一些铺垫。

Windows 程序的一项主要内容是打开窗口。窗口提供了用于显示文本和图形的工作空间，让用户能够同应用程序交互。创建完整 Windows 程序的步骤如下：

- 创建一个 Windows 类；
- 创建一个事件处理程序（WinProc）；
- 向 Windows 注册创建的 Windows 类；
- 使用 Windows 类创建一个窗口；
- 创建一个主事件循环，用于接收 Windows 消息并将其发送给事件处理程序。

下面详细介绍每个步骤。

2.3.1　Windows 类

Windows 是一个真正的面向对象的操作系统，因此在 Windows 中的很多概念和过程都源自 C 和 C++，Windows 类就是其中之一。在 Windows 中，窗口、控件、列表框、对话框等实质上都是窗口，它们之间的差别在于定义它们的类。每个 Windows 类都是一种 Windows 可以处理的窗口类型描述。

在 Windows 中有大量预定义的 Windows 类，如按钮、列表框、文件选择器等。但是，也可以很方便地创建自己的 Windows 类。实际上，对于每个应用程序，至少需要创建一个 Windows 类；否则，程序将相当复杂。因此可以将 Windows 类视为模板，使用它可以生成窗口，并处理消息。

有两种数据结构可用于保存 Windows 类消息：WNDCLASS 和 WNDCLASSEX。WNDCLASS 是一种较早的数据结构，可能很快会废弃不用，因此我们将使用新的扩展版本 WNDCLASSEX。这两种数据结构极其类似，如果您对它们感兴趣，可在 Win32 帮助中查找到旧版本的 WNDCLASS。下面是 Windows 头文件中定义的 WNDCLASSEX 数据结构：

```
typedef struct _WNDCLASSEX
    {
    UINT  cbSize;     // 该结构的大小
    UINT  style;      // 样式标记
    WNDPROC lpfnWndProc;  // 指向事件处理程序的函数指针
    int   cbClsExtra; // 额外的类信息
    int   cbWndExtra;   // 额外的窗口信息
    HANDLE hInstance;   // 应用程序实例
    HICON  hIcon;     // 主图标
    HCURSOR hCursor;      // 鼠标
    HBRUSH hbrBackground; // 用于绘制窗口的背景刷
    LPCTSTR lpszMenuName; // 菜单名称
    LPCTSTR lpszClassName; // 类名
    HICON  hIconSm;     // 小图标句柄
    } WNDCLASSEX;
```

您需做的是创建一个这样的数据结构，并对其各个字段赋值。

```
WNDCLASSEX winclass; //  一个空的 windows 类
```

下面介绍如何对各个字段赋值。

第一个字段 cbSize 非常重要。它指的是结构 WNDCLASSEX 本身的大小。您可能会问，为什么数据结构需要知道自己的大小呢？因为将结构作为指针进行传递时，接收方可通过检查第一个字段（4 字节）来判断数据块的大小，然后直接跳到结构的末尾。它提供了一些帮助信息，让其他函数无需在运行时计算类的大小。因此，您需要设置该字段：

```
winclass.cbSize = sizeof(WNDCLASSEX);
```

接下来的字段是样式标记，描述窗口的常规属性。这些标记有很多，这里不打算全部介绍，您只需要知道使用这些标记可以创建任何类型的窗口。表 2.5 列出了一些最常用的标记，可以使用逻辑 OR 来将这些值组合起来，以设置所需的窗口类型。

表 2.5　　Windows 类的样式标记

标记	含义
CS_HREDRAW	移动窗口或调整窗口宽度时重新绘制整个窗口
CS_VREDRAW	移动窗口或调整窗口高度时重新绘制整个窗口
CS_OWNDC	为类中的每个窗口分配一个惟一的设备场境
CS_DBLCLKS	当用户在窗口内双击鼠标时，发送双击消息给窗口过程
CS_PARENTDC	设置子窗口与其父窗口的重叠区域，以便在父窗口上绘制子窗口
CS_SAVEBITS	存储窗口中的客户区域图像（client image），以免每当窗口被遮掩、移动时您不需要重绘窗口。但这样做将占用更多的内存，速度也更慢
CS_NOCLOSE	禁用 System 菜单上的 Close 命令

表 2.5 中的标记很多，很容易让人混淆。就现在而言，我们将这样设置样式标记，即让窗口被移动或调整大小时重绘，使用静态设备场境，能够处理鼠标双击事件。

设备场境（Device Context）用于将 GDI 图形渲染到窗口中。因此要处理图形，需要为窗口请求设备场境。如果使用标记 CS_OWNDC，将 Windows 类设置为自己的设备场境，可节省一些时间，这样可避免每次都请求设备场境。下面的代码说明了如何设置 style 字段，使窗口满足我们的需求：

```
winclass.style = CS_VREDRAW | CS_HREDRAW | CS_OWNDC | CS_DBLCLICKS;
```

WNDCLASSEX 结构的下一个字段 lpfnWndProc 是一个函数指针，它指向一个事件处理程序。通常，将这个字段设置类的回调函数。回调函数在 Windows 编程中很常见，其工作原理如下：发生事件时，不需要进行随机轮询，Windows 将通过调用回调函数来通知您。在回调函数内部，您可以采取任何措施。

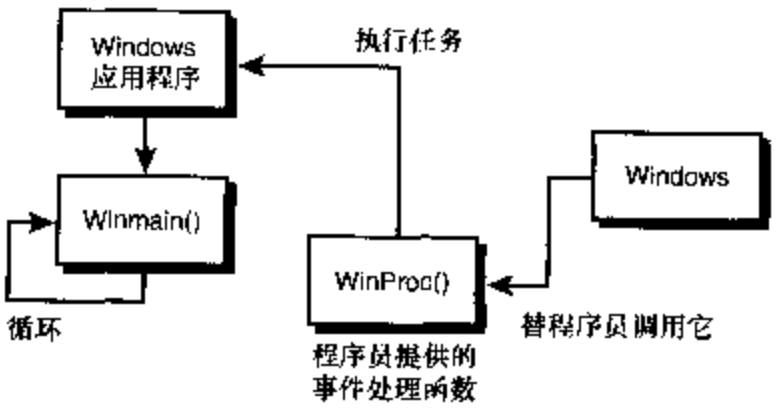


图 2.3　Windows 事件处理函数

这说明了基本的 Windows 事件循环和事件处理程序的工作原理。您给 Windows 类提供回调函数，发生事件时，Windows 将为您调用该函数，如图 2.3 所示。本章后面将更详细地介绍这一点；现在，只需将其设置为我们将编写的事件函数。

```
winclass.lpfnWndProc = WinProc; // 这是我们将编写的函数
```

注意：您可能不熟悉函数指针，它们类似于 C++中的虚函数。您可能也不熟悉虚函数，则有必要解释一下。假定有两个函数，用于对两个数字执行运算：

```
int Add(int op1, int op2) {return(op1+op2);}
```

```
int Sub(int op1, int op2) {return(op1-op2);}
```

要想用相同的调用方式调用其中任何一个函数，可以使用函数指针：

```
// 定义一个函数指针，该指针可指向这样的函数，即接受两个 int 参数并返回一个 int 值：
int (Math*)(int, int);
```

然后像下面这样给函数指针赋值：

```
Math = Add;
int result = Math(1,2); // 相当于调用 Add(1,2)
// 结果为 3

Math = Sub;
int result = Math(1,2); // 相当于调用 Sub(1,2)
// 结果为-1
```

接下来的两个字段 cbClsExtra 和 cbWndExtra，最初设计用来指示 Windows 在 Windows 类中保留一些额外空间，以保存运行时信息。但是，现在大多数人都不使用这两个字段，而是简单地将它们设置为 0：

```
winclass.cbClsExtra = 0; // 额外的类信息空间
winclass.cbWndExtra = 0; // 额外的窗口信息空间
```

下一个字段是 hInstance。这个字段是程序启动时传递给 WinMain()函数的 hinstance，因此只需要从 WinMain()中复制它即可：

```
winclass.hInstance = hinstance; // 将其设置为应用程序实例
```

其他的字段与 Windows 类的图形方面相关，介绍它们之前，先快速回顾一下句柄。

在 Windows 程序和类型中，您将经常看到句柄：位图句柄、光标句柄等。别忘了，句柄是基于内部 Windows 类型的标识符。实际上它们是整数，但 Microsoft 可能改变这种做法，因此最好使用 MS 类型。您将看到越来越多的“空白句柄（handles to blank）”，任何以 h 为打头的类型通常都是句柄。

下一个字段用于设置代表应用程序的图标类型。您可以加载自己定制的图标，但我们现在将使用系统图标——您需要一个图标句柄。要获得一个指向通用系统图标的句柄，可使用函数 LoadIcon()：

```
winclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
```

上述代码加载标准应用程序图标：虽然单调，但很简单。如果您对 LoadIcon()函数感兴趣，可查看 Win32 API 文档——其中有大量预定义图标可供使用。

接下来的字段是 hCursor。与 hIcon 类似，它也是一个图形对象句柄；但不同的是，hCursor 是一个光标句柄，鼠标指针进入窗口的客户区域后，将显示 hCursor 指向的光标，它代表鼠标指针形状。函数 LoadCursor()用于获得到资源光标或预定义系统光标的句柄。资源将在后面介绍，它们是编译到应用程序中的数据，如位图、光标、图标、声音等，在运行时可访问它们。下面的代码将 Windows 类的光标设置为标准箭头：

```
winclass.hCursor = LoadCursor(NULL, IDC_ARROW);
```

同样，如果您对光标样式感兴趣，可以在 Win32 API 中查找相关信息。

接下来的字段是 hbrBackground。重新绘制或刷新窗口时，Windows 至少会使用预定义的颜色（在 Windows 中称为“画刷”）重新绘制窗口客户区域的背景。hbrBackground 是用于刷新窗口的画刷句柄。画刷、画笔、颜色和图形都是图形设备接口（GDI，Graphics Device Interface）的组成部分，我们不必关心 GDI，因为我们主要使用 DirectX。下面演示如何请求基本的系统画刷，将其用于绘制窗口。这是通过函数 GetStockObject()来完成的，如下列代码行所示：

```
winclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
```

警告：注意强制类型转换（HBRUSH）。对于 Visual C++ 6.0+编译器，它是必需的！对于其他编译器，使用该类型转换也没有坏处。

GetStockObject()函数是一个通用函数，它可用于获取 Windows“库存”的画刷、画笔、调色板或字体句柄。GetStockObject()接受一个参数，指出要加载哪种资源。表 2.6 列出了可用的画刷和画笔对象。

表 2.6 GetStockObject( )的对象标识符

值	含义	值	含义
BLACK_BRUSH	黑色画刷	HOLLOW_BRUSH	中空画刷
WHITE_BRUSH	白色画刷	NULL_BRUSH	空画刷
GRAY_BRUSH	灰色画刷	BLACK_PEN	黑色画笔
LTGRAY_BRUSH	淡灰色画刷	WHITE_PEN	白色画笔
DKGRAY_BRUSH	深灰色画刷	NULL_PEN	空画笔

大多数情况下，窗口的背景画刷都无关紧要，因为在 DirectX 接管之后，它将不起作用。

WNDCLASS 结构中接下来的字段是菜单字段 lpszMenuName。它是一个以 NULL 结尾的 ASCII 字符串，包含要加载并附加到窗口上的菜单资源的名称。因为现在我们不使用菜单，所以将它设置为 NULL：

```
winclass.lpszMenuName = NULL; // 要加入到窗口中的菜单的名称
```

如前所述，每个 Windows 类代表应用程序可以创建的一种窗口类型。类与一般意义上的模板类似；因此，Windows 需要跟踪并识别它们，字段 lpszClassName 就是用于这个目的。该字段是一个以 NULL 结尾的 ASCII 字符串，包含类的文本标识符。一般来说，我个人使用诸如 WINCLASS1、WINCLASS2 之类的标识符。使用什么样的类名称取决于您，但最好使它尽可能简单：

```
winclass.lpszClassName = "WINCLASS1"; // 类本身的名称
```

进行上述赋值后，就可以用类名称来引用新的 Windows 类，如 WINCLASS1。

最后一个字段是“小”应用程序图标。这是 Windows 类 WNDCLASSEX 新增加的一个字段，在以前的 WNDCLASS 类中没有这个字段。实际上，该句柄指向窗口的标题栏和 Windows 桌面任务栏显示的图标。通常加载一个自定义资源，但现在我们使用 LoadIcon()函数来加载一个标准 Windows 图标：

```
winclass.hIconsm = LoadIcon(NULL, IDI_APPLICATION);
```

现在来看一看整个类定义：

```
WNDCLASSEX winclass; // 要创建的窗口类

// 设置窗口类
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style   = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc = WindowProc;
winclass.cbClsExtra = 0;
winclass.cbWndExtra = 0;
winclass.hInstance  = hinstance;
winclass.hIcon      = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor    = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL;
winclass.lpszClassName  = "WINCLASS1";
winclass.hIconSm    = LoadIcon(NULL, IDI_APPLICATION);
```

当然，要减少输入量，可在声明窗口类的同时进行初始化，如下所示：

```
WNDCLASSEX winclass = {
        winclass.cbSize = sizeof(WNDCLASSEX),
        CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW,
        WindowProc,
        0,
        0,
        hinstance,
        LoadIcon(NULL, IDI_APPLICATION),
        LoadCursor(NULL, IDC_ARROW),
        (HBRUSH)GetStockObject(BLACK_BRUSH),
        NULL,
        "WINCLASS1",
        LoadIcon(NULL, IDI_APPLICATION)};
```

2.3.2　注册 Windows 类

定义 Windows 类并将其存储到 winclass 中后，还必须通过注册让 Windows 操作系统知道这个类。注册是通过函数 RegisterClassEx()来完成的，它只接受一个指向新类的指针作为参数：

```
RegisterClassEx(&winclass);
```

警告：这里没有使用类名（WINCLASS1）。对于 RegisterClassEx( )函数，您必须使用存储类的实际数据结构，因为调用该函数之前，Windows 系统还不知道这个类，因此不能使用类名 WINCLASS1 来引用它。

另外，还有一个老式函数 RegisterClass()，它用于注册基于数据结构 WNDCLASS 的类。

注册类后，可以使用它来创建窗口。下面介绍如何创建窗口，然后重温事件处理程序和主事件循环的细节，让读者知道为使 Windows 应用程序正常运行，需要进行哪些处理。

2.3.3　创建窗口

可以用函数 CreateWindow()或 CreateWindowEx()来创建窗口（或任何其他类似于窗口的对象，如控件）。后一个函数是最新的扩展，支持更多的样式参数，我们将使用它。创建窗口时，必须提供窗口类的文本名称，这里为 WINCLASS1。它用于标识窗口类，将其同其他窗口类和内嵌类型（如按钮、文本框等）区分开来。

下面是 CreateWindowEx()的函数原型：

```
HWND CreateWindowEx(
   DWORD dwExStyle,   // 扩展的窗口样式
   LPCTSTR lpClassName, // 类名指针
   LPCTSTR lpWindowName, // 窗口名指针
   DWORD dwStyle,     // 窗口样式
   int x,          // 窗口的水平位置
   int y,          // 窗口的垂直位置
   int nWidth,      // 窗口宽度
   int nHeight,     // 窗口高度
   HWND hWndParent,   // 父窗口句柄
   HMENU hMenu,      // 菜单句柄或子窗口标识符
   HINSTANCE hInstance, // 应用程序实例句柄
   LPVOID lpParam);   // 指向窗口创建数据的指针
```

如果该函数成功运行，将返回一个指向新建窗口的句柄；否则返回 NULL。

该函数中大多数参数的含义都是不言自明的，这里还是解释一下：

- dwExStyle——该扩展样式标记是一种高级功能，大多数情况下都被设置为 NULL。要了解该标记的各种可能值，可查看 Win32 SDK 帮助文档。我曾经使用过的惟一一个标记值是 WS_EX_TOPMOST，它确保窗口始终位于最上面。
- lpClassName——该参数指定使用哪个类来创建窗口，如 WINCLASS1。
- lpWindowName：它是一个以 NULL 结尾的字符串，包含窗口的标题，如“My First Window”。
- dwStyle——它是常规窗口标记，用于指定窗口的外观和行为，是一个非常重要的参数。表 2.7 列出了对于 DirectX 应用程序较常用的一些值。当然，可以使用逻辑 OR 将这些值组合起来，以获得所需的功能。
- *x*、*y*——以像素坐标指定窗口的左上角位置。如果您不关心窗口的位置，可使用 CW_USEDEFAULT，Windows 将自动确定窗口的位置。
- nWidth、nHeight——指定窗口的宽度和高度，单位为像素。如果您不关心窗口的大小，可使用 CW_USEDEFAULT，Windows 将自动确定窗口的大小。
- hWndParent——指向父窗口（如果存在）的句柄。如果不存在父窗口，可使用 NULL；在这种情况下，桌面将是其父窗口。
- hMenu——附加到窗口上的菜单的句柄，下一章将更详细介绍。现在只需使用 NULL 即可。
- hInstance——应用程序实例，这里使用 WinMain()函数中的 hinstance。
- lpParam——高级参数。设置为 NULL。

表 2.7　　dwStyle 的常规样式值

样式	含义
WS_POPUP	创建一个弹出式窗口
WS_OVERLAPPED	创建一个重叠窗口。重叠窗口有标题栏和边框，与 WS_TILED 样式相同
WS_OVERLAPPEDWINDOW	使用样式 WS_OVERLAPPED、WS_CAPTION、WS_SYSMENU、WS_THICKFRAME、WS_MINIMIZEBOX 和 WS_MAXIMIZEBOX 创建一个重叠窗口
WS_VISIBLE	创建一个一开始就可见的窗口

下面的代码创建一个这样的基本重叠窗口，即包含标准控件、位于(0，0)、大小为 400×400 像素的窗口：

```
HWND hwnd; // 窗口句柄

// 创建窗口
if (!(hwnd = CreateWindowEx(NULL, // 扩展样式
                "WINCLASS1",      // 窗口类
           "Your Basic Window", // 标题
          WS_OVERLAPPEDWINDOW | WS_VISIBLE,
           0,0,    // 初始位置
          400,400, // 初始宽度和高度
          NULL,    // 父窗口句柄
          NULL,    // 菜单句柄
          hinstance,// 应用程序实例
          NULL)))  // 其他创建参数
return(0);
```

窗口被创建后，其状态为可见或不可见。但这里使用了样式标记 WS_VISIBLE，它使窗口自动可见；如果没有该标记，可使用下列函数调用手动显示窗口：

```
// 显示窗口
ShowWindow(hwnd, ncmdshow);
```

还记得 WinMain() 函数中的参数 ncmdshow 吗？它是一个非常方便的参数。尽管可以使用 WS_VISIBLE 标记覆盖它，但通常将它作为参数传送给 ShowWindow() 函数。接下来您可能想强制 Windows 更新窗口的内容并生成一个 WM_PAINT 消息。这可通过调用函数 UpdateWindow() 来完成：

```
// 将一条 WM_PAINT 发送给窗口，以刷新其内容
UpdateWindow();
```

2.3.4　事件处理程序

下面来看一下主事件处理程序。前面介绍过，事件处理程序是一个回调函数，Windows 在窗口中发生需要处理的事件时，在主事件循环中调用该函数。关于回调函数的流程，可参见图 2.3。

事件处理程序是由您编写的，因此可处理任何想处理的事件；其他的事件可传递给 Windows 进行处理。当然，应用程序处理的事件和消息越多，其功能越强。

编写代码之前，先介绍一下事件处理程序的细节、其功能和工作原理。首先，对于创建的每个 Windows 类，都可以拥有独立的事件处理程序，我称它为 Windows 过程（以下简称为 WinProc）。主事件循环从用户或 Windows 那里收到消息，在将其加入到主事件队列中时，将把它发送给 WinProc。这些说起来有些复杂，所以我将以另一种方式解释。

用户和 Windows 执行某些操作时，将生成一些针对应用程序窗口的事件和消息。所有这些消息都将进入一个队列中，但针对您的应用程序窗口的事件和消息将进入到该窗口的私有队列中。然后主事件循环将获取到这些消息，并将它们发送给窗口的 WinProc 进行处理，如图 2.1 所示。

实际上，有数以百计的事件和消息，本书不会全部介绍它们，幸运的是，只需要处理很少几个消息，就可以让 Windows 应用程序正常启动和运行。

简单地说，主事件循环将事件和消息提供给 WinProc，然后 WinProc 对它们进行处理。因此，不仅我们需要关心 WinProc，主事件循环也需要考虑它，这将稍后介绍。知道 WinProc 的功能后，来看看它的函数原型：

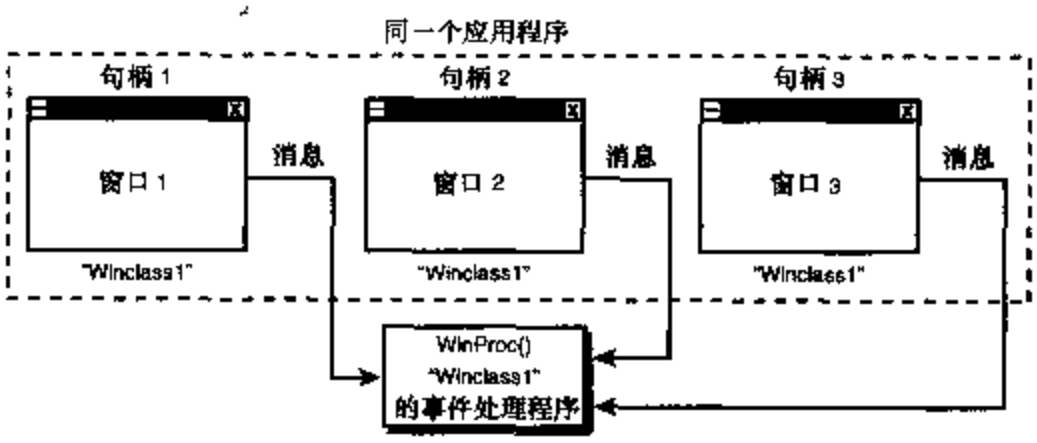
```
LRESULT CALLBACK WindowProc(
        HWND hwnd, // 发送方的窗口句柄
        UINT msg, // 消息 id
        WPARAM wparam, // 更详细地定义消息
        LPARAM lparam); // 更详细地定义消息
```

当然，这只是回调函数的原型。您可以调用任何函数，只需将函数的地址作为函数指针赋给 winclass.lpfnWndProc 即可：

```
winclass.lpfnWndProc = WindowProc;
```

这些参数的含义是不言自明的：

- hwnd——它是窗口句柄。该参数只有当您使用同一个 Windows 类打开了多个窗口时有意义。这时只有使用 hwnd 才能分辨出消息来自哪个窗口。图 2.4 说明了这种情况。
- msg——它是 WinProc 应处理的实际消息 ID。该 ID 可以是多个主要消息之一。
- wparam 和 lparam——这两个参数用于进一步确定参数 msg 指定的消息。



基于同一个类的所有窗口的消息
都由同一个事件处理程序进行处理

图 2.4　基于同一个类的多个窗口

- 最后，返回类型和声明符号也相当重要：LRESULT CALLBACK。这些关键字是必不可少的，不要遗漏它们！

大多数人的做法是，使用 switch() 来判断 msg，为每种情况编写相应的代码。根据 msg 的内容，可以知道是否需要进一步查看 wparam 和 lparam 的值。表 2.8 列出了一些基本的消息 ID。

表 2.8　　部分消息 ID

值	含义	值	含义
WM_ACTIVATE	窗口被激活或获取焦点时发送该消息	WM_KEYDOWN	按下键时发送该消息
WM_CLOSE	关闭窗口时发送该消息	WM_TIMER	发生计时器事件时发送该消息
WM_CREATE	首次创建窗口时发送该消息	WM_USER	让您能够发送消息
WM_DESTROY	要销毁窗口时发送该消息	**WM_PAINT**	窗口需要重绘时发送该消息
WM_MOVE	移动窗口时发送该消息	**WM_QUIT**	Windows 应用程序终止时发送该消息
WM_MOUSERMOVE	移动鼠标时发送该消息	WM_SIZE	窗口改变大小时发送该消息
WM_KEYUP	松开键时发送该消息		

注：用粗体显示的是最常用的消息。

通常，应用程序运行时，将发送一个或多个消息给 WinProc 进行处理。消息 ID 位于 msg 中，其他信息存储在 wparam 和 lparam 中。要了解消息有哪些参数，最好参考 Win32 SDK 联机帮助。

幸运的是，我们现在只关心三种消息：

- WM_CREATE——最初创建窗口时发送该消息，它提供一个进行设置、初始化或资源分配的机会。

● WM_PAINT——窗口内容需要重绘时发送该消息。它可能由很多种原因导致：用户移动窗口或调整窗口的大小、被另一个应用程序弹出窗口遮住等。

● WM_DESTROY——窗口要被销毁时发送该消息。通常是用户单击了窗口的关闭图标或从窗口的系统菜单中选择关闭后的直接结果。这时，您需要释放所有资源，并通过发送 WM_QUIT 消息，告诉 Windows 完全终止该应用程序。

下面来看一个用于处理所有这些消息的完整 WinProc，如下列代码所示：

```
LRESULT CALLBACK WindowProc(HWND hwnd,
                            UINT msg,
                            WPARAM wparam,
                            LPARAM lparam)
{
// 这是系统的主消息处理函数
PAINTSTRUCT  ps;  // 用于 WM_PAINT
HDC     hdc;  // 设备场境句柄

// 判断消息是什么
switch(msg)
  {
  case WM_CREATE:
    {
  // 执行初始化的代码

     // 成功返回
  return(0);
  } break;

  case WM_PAINT:
  {
  // 确认窗口是否有效
  hdc = BeginPaint(hwnd,&ps);
  // 执行绘制的代码
    EndPaint(hwnd,&ps);
    // 成功返回
  return(0);
  } break;

  case WM_DESTROY:
  {
  // 关闭应用程序，下述代码发送一条 WM_QUIT 消息
  PostQuitMessage(0);

     // 成功返回
  return(0);
  } break;

  default:break;

  } // end switch

// 处理其他消息
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc
```

首先来看一下对消息 WM_CREATE 的处理，这里只是 return(0)，它告诉 Windows 已经对该消息进行了处理，不要再采取措施。当然，也可以在处理 WM_CREATE 消息时执行各种的初始化操作。

下一个消息是 WM_PAINT，它非常重要。窗口需要重新绘制时发送该消息，这意味着大多数时候您需要执行重绘操作。对于 DirectX 游戏，这无关紧要，因为我们将以 30～60fps（帧/秒）的速度重绘屏幕，但对于普通 Windows 应用程序而言，这就很重要了。然而，对于 DirectX 应用程序，我们不希望 Windows 认为屏幕没有被重新绘制，所以需要让 Windows 知道我们已经对 WM_PAINT 消息进行处理了。

为此，必须“验证窗口的客户矩形”。有很多种方法可以完成这种处理，但最简单的方法是调用函数

BeginPaint() 和 EndPaint()。这两个函数验证窗口，并用存储在 Windows 类变量 hbrBackground 中的背景画刷重新填充背景。下面是用于验证的代码：

```
  // 开始绘制
hdc = BeginPaint(hwnd,&ps);
// 完成绘制工作的代码
 EndPaint(hwnd,&ps);
```

有两点需要注意。首先，每个函数的第一个参数都是窗口句柄 hwnd。它是必不可少的，因为函数 BeginPaint()/EndPaint() 可能在应用程序的任何一个窗口中进行绘图；因此需要使用窗口句柄指定要在哪个窗口中进行绘图。第两个参数是 PAINTSTRUCT 结构的地址，该结构包含了要重绘的区域。也就是说，可能只有一个小型区域需要重绘，因此使用该结构来指定这个区域。下面是该结构的定义：

```
typedef struct tagPAINTSTRUCT
    {
    HDC hdc;
    BOOL fErase;
    RECT rcPaint;
    BOOL fRestore;
    BOOL fIncUpdate;
    BYTE rgbReserved[32];
    } PAINTSTRUCT;
```

讨论 GDI 之前，读者不必关心该结构中的大部分字段，但有一个最重要的字段——rcPaint，它是一个 RECT 结构，包含需要重绘的最小矩形，如图 2.5 所示。

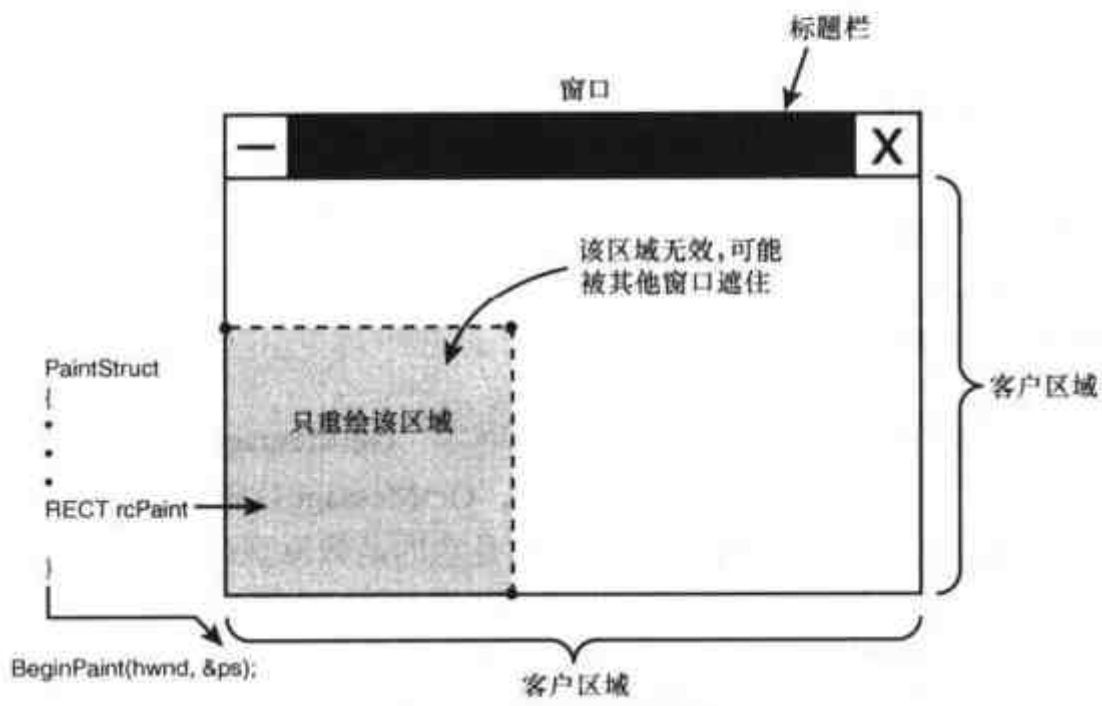


图 2.5　只重绘无效区域

Windows 试图做尽可能少的工作，因此当窗口的内容遭到破坏时，Windows 试图告诉您需要重新绘图以恢复窗口内容的最小矩形区域，如下列代码所示：

```
typedef struct tagRECT
    {
    LONG left;   // 矩形左上角的 x 坐标
    LONG top;  // 矩形左上角的 y 坐标
    LONG right; // 矩形右下角的 x 坐标
    LONG bottom; // 矩形右下角的 y 坐标
    } RECT;
```

关于函数调用 BeginPaint()，需要注意的最后一点是，它返回一个图形场境句柄：

```
HDC hdc; // 图形场境句柄
hdc = BeginPaint(hwnd,&ps);
```

图形场境（graphic context）是一种用于描述视频系统和绘图面的数据结构。要操作图形，需要获得一个图形场境。对消息 WM_PAINT，暂时就介绍这么多。

WM_DESTROY 消息实际上是需要特别注意的。用户关闭窗口时将发送这种消息。但是，它只关闭窗口，而不会关闭应用程序。应用程序将继续运行，但是没有窗口。所以，在这时您需要做一些事情。大多数情况下，用户关闭程序的主窗口时，实际上他是想终止应用程序。因此，您必须发送一条退出消息，帮助终止应用程序，该消息为 WM_QUIT。由于它是一条很常用的消息，所以有一个 PostQuitMessage() 函数专门用于发送它。

在 WM_DESTROY 处理程序中，需要执行所有的清理工作，然后调用函数 PostQuitMessage(0)，命令 Windows 终止应用程序。这将把 WM_QUIT 消息加入消息队列中，它将导致主事件循环结束。

在 WinProc 事件处理程序中，有几个细节需要注意。首先，读者可能注意到了每个处理程序体后面的 return(0) 语句。它有两个用途：退出 WinProc；告诉 Windows 已经对消息进行了处理。第二个重要的细节是，默认消息处理程序 DefaultWindowProc()。该函数是一个传递函数，将未处理的消息传递给 Windows 进行默认处理。因此，如果没有处理某条消息，应在事件处理程序函数末尾使用如下的函数调用：

```
// 传递未处理的消息
return (DefWindowProc(hwnd, msg, wparam, lparam));
```

这看似多余且得不偿失，但有了基本的 Windows 应用程序框架后，只需复制它并添加自己的代码。实际上，设置好虚拟计算机后，我们就可以使用它，而无需考虑任何与 Windows 相关的内容。下面来看一看主事件循环。

2.3.5 主事件循环

主事件循环相当简单，如下所示：

```
// 进入主事件循环
while(GetMessage(&msg,NULL,0,0))
   {
   // 转换加速键
   TranslateMessage(&msg);
   // 将消息发送给 window proc
   DispatchMessage(&msg);
   } // end while
```

主循环 while() 将一直执行，直到 GetMessage() 返回零。GetMessage() 是主事件循环的核心，它惟一的用途是从事件队列中获得下一条消息并对其进行处理。GetMessage() 函数有 4 个参数，第一个参数对于我们来说很重要，其他参数可设置为 NULL 或零。下面是它的函数原型：

```
BOOL GetMessage(
   LPMSG lpMsg,     // address of structure with message
   HWND hWnd,       // handle of window
   UINT wMsgFilterMin, // first message
   UINT wMsgFilterMax); // last message
```

参数 msg 是 Windows 用于放置文本消息的存储空间。但是，与 WinProc() 函数的 msg 参数不同，这里的 msg 是一个复杂的数据结构，而不是一个整数 ID。消息进入 WinProc 后，将被划分成多个组成部分。下面是 MSG 结构：

```
typedef struct tagMSG
    {
    HWND hwnd;   // 发生事件的窗口
    UINT message; // 消息 ID
    WPARAM wParam; // 更详细的消息信息
    LPARAM lParam; //更详细的消息信息
    DWORD time;  // 事件发生的时间
    POINT pt;   // 鼠标位置
    } MSG;
```

这个数据结构包含了 WinProc() 的所有参数以及其他一些参数，如事件发生时的时间和鼠标位置。

我们知道，GetMessage() 从事件队列中获取下一条消息，但是之后呢？接下来将调用 TranslateMessage()

函数。TranslateMessage()是一个虚拟加速键转换器。您只需调用它，而不用关心它做什么。在最后一个函数 DispatchMessage()中将执行要采取的措施。GetMessage()从队列中获取消息，并由 TranslateMessage()对其进行处理和转换后，将通过调用 DispatchMessage()函数，来调用 WinProc()。

DispatchMessage()将调用 WinProc，并根据原始数据结构 MSG 发送适当的参数。图 2.6 说明了主事件循环的整个过程。

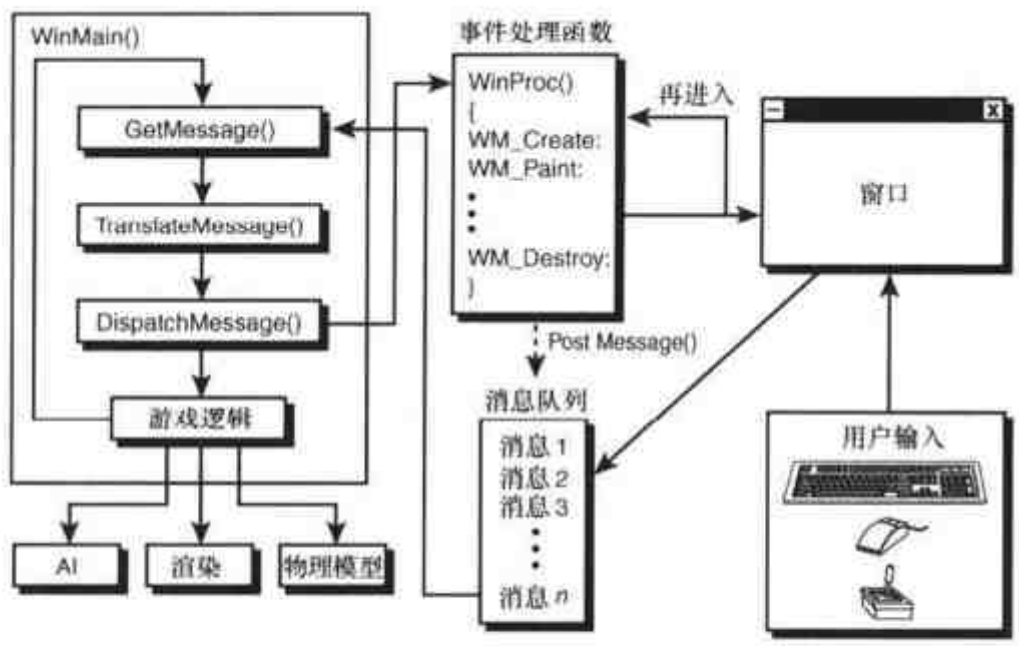


图 2.6　事件循环消息处理机制

这就是全部内容，现在您已成为一名 Windows 专家！余下的就是细节了，但如果您掌握了前面介绍的概念，并知道了事件循环、事件处理程序等的重要性，便离成功不远了。来看一下下面的代码清单，这是一个完整的 Windows 程序，它创建一个窗口，并等待用户关闭它。

```
// DEMOII2_3.CPP：一个完整的 Windows 程序

// INCLUDES ///////////////////////////////////////////////
#define WIN32_LEAN_AND_MEAN  // 不使用 MFC

#include <windows.h>  // 包含所有的 Windows 头文件
#include <windowsx.h> // 包含有用的宏
#include <stdio.h>
#include <math.h>

// 常量 ///////////////////////////////////////////////////

// 用于窗口的常量
#define WINDOW_CLASS_NAME "WINCLASS1"

// 全局变量 ///////////////////////////////////////////////


// 函数 ///////////////////////////////////////////////////
LRESULT CALLBACK WindowProc(HWND hwnd,
        UINT msg,
             WPARAM wparam,
             LPARAM lparam)
{
// 这是系统的主消息处理函数
PAINTSTRUCT  ps; // used in WM_PAINT
HDC     hdc; // 设备场境句柄

// 检测是什么消息
switch(msg)
  {
  case WM_CREATE:
    {
  // 初始化代码
```

```
    // 成功返回
  return(0);
  } break;

  case WM_PAINT:
  {
  // 验证窗口是否有效
  hdc = BeginPaint(hwnd,&ps);
  // 绘制代码
    EndPaint(hwnd,&ps);

    // return success
  return(0);
  } break;

  case WM_DESTROY:
  {
  // 关闭应用程序，这将发送一条 WM_QUIT 消息
  PostQuitMessage(0);

    // 成功返回
  return(0);
  } break;

  default:break;

  } // end switch

// 处理其他消息
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////////
int WINAPI WinMain( HINSTANCE hinstance,
      HINSTANCE hprevinstance,
      LPSTR lpcmdline,
      int ncmdshow)
{

WNDCLASSEX winclass; // 用于存储创建的窗口类
HWND    hwnd;   // 窗口句柄
MSG    msg;     // 消息

// 设置窗口类
winclass.cbSize  = sizeof(WNDCLASSEX);
winclass.style    = CS_DBLCLKS | CS_OWNDC |
         CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc   = WindowProc;
winclass.cbClsExtra   = 0;
winclass.cbWndExtra   = 0;
winclass.hInstance    = hinstance;
winclass.hIcon     = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor   = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground   = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName   = NULL;
winclass.lpszClassName   = WINDOW_CLASS_NAME;
winclass.hIconSm    = LoadIcon(NULL, IDI_APPLICATION);

// 注册窗口类
if (!RegisterClassEx(&winclass))
  return(0);

// 创建窗口
if (!(hwnd = CreateWindowEx(NULL, // 扩展样式
              WINDOW_CLASS_NAME,   // 窗口类
```

```
            "Your Basic Window", // 标题
            WS_OVERLAPPEDWINDOW | WS_VISIBLE,
             0,0,     // 窗口的初始位置
            400,400,    // 初始宽度和高度
            NULL,      // 父窗口句柄
            NULL,      // 菜单句柄
            hinstance,// 应用程序实例
            NULL)))     // 额外的创建参数
    return(0);

    // 进入主循环
    while(GetMessage(&msg,NULL,0,0))
       {
       // 转换加速键
       TranslateMessage(&msg);

       // 将消息发送给 Windowproc
       DispatchMessage(&msg);
       } // end while

    // 返回到 Windows 操作系统
    return(msg.wParam);

    } // end WinMain

    ///////////////////////////////////////////////////////////////
```

要编译 DEMOII2_3.CPP，只需创建一个 Win32 .EXE 应用程序，并将 DEMOII2_3.CPP 加入到该工程中。也可以从附带光盘中运行编译好的 DEMOII2_3.EXE。图 2.7 是该程序运行时的屏幕截图。



图 2.7　DEMOII2_3.EXE 运行时的屏幕截图

继续介绍后面的内容之前，还有两点需要说明。首先，如果您仔细查看事件循环，将发现它不是实时的：程序通过 GetMessage() 函数等待消息时，主事件循环实际上被阻断。确实是这样，必须想办法解决这个问题，因为我们需要连续地执行游戏处理，并在 Windows 事件发生时及时处理它。

2.3.6　构建实时事件循环

实时非等待/非阻断事件循环是非常容易构建的。只需要一种检测消息队列中是否有消息的方法。如果有，对其进行处理；否则，继续处理其他游戏逻辑并重复循环。用于执行测试功能的函数为 PeekMessage()，其函数原型与 GetMessage() 非常类似，如下所示：

```
BOOL PeekMessage(
   LPMSG lpMsg,    // 消息结构指针
   HWND hWnd,      // 窗口句柄
   UINT wMsgFilterMin, // 第一条消息
   UINT wMsgFilterMax, // 最后一条消息
   UINT wRemoveMsg);  // 删除标记
```

消息队列中有消息时，它将返回一个非零值。

差别在于最后一个参数。该参数指定如何从消息队列中获取消息。参数 wRemoveMsg 的有效标记如下：

- PM_NOREMOVE——消息被 PeekMessage() 处理之后，不将其从消息队列中删除。
- PM_REMOVE——消息被 PeekMessage() 处理后，将其从消息队列中删除。

考虑到这两个可能取值，可以按两种方法来处理消息：使用 PeekMessage() 和 PM_NOREMOVE 标记，如果消息队列中有消息，则调用 GetMessage() 函数；使用 RM_REMOVE 标记，并使用 PeekMessage() 本身来获取消息。我们采用后一种方法。下面是核心逻辑代码部分，对其进行了修改，以便在主事件循环中

使用刚才介绍的新技术：

```
while(TRUE)
  {
  // 检测消息队列中是否有消息，如果有，读取它
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // 检测是否是退出消息
     if (msg.message == WM_QUIT)
      break;

    // 转换加速键
    TranslateMessage(&msg);

    // 将消息发送给 window proc
    DispatchMessage(&msg);
    } // end if

    // 主游戏处理逻辑
  Game_Main();
  } // end while
```

代码中最重要的部分为粗体。第一个重要部分如下：

```
if (msg.message == WM_QUIT)
  break;
```

上述代码说明了如何退出无限循环 while（TRUE）。在 WinProc 中处理消息 WM_DESTROY 时，必须需要调用 PostQuitMessage()来发送一条 WM_QUIT 消息。这样 WM_QUIT 消息将进入消息队列，进而我们可以检测到该消息并退出循环。

第二个重要部分指出了应在哪里调用主游戏循环。函数调用 Game_Main()在生成一个动画帧或执行游戏逻辑后必须返回；否则，主窗口事件循环将无法处理消息。

为演示这种新的实时结构更适合用于游戏逻辑处理，请看附带光盘中的 DEMOII2_4.CPP 和 DEMOII2_4.EXE。这种结构实际上就是本书后面将介绍的虚拟计算机使用的模型。

读者对基本的 Windows 编程有一定了解后，接下来介绍游戏中需要使用的图形、声音、和输入接口：DirectX。

2.4 DirectX 和 COM 简明教程

DirectX 是一个软件系统，它抽象化了图形、声音、输入、网络、安装等，因此不管 PC 的硬件配置如何，都可以使用相同的代码。此外，DirectX 技术与 Windows 固有的 GDI 和 MCI（Media Control Interface，媒体控制接口）相比，速度要快很多倍，且更健壮。图 2.8 说明了使用和不使用 DirectX 时如何编写 Windows 游戏。从中可知，DirectX 解决方案相当简洁优美。

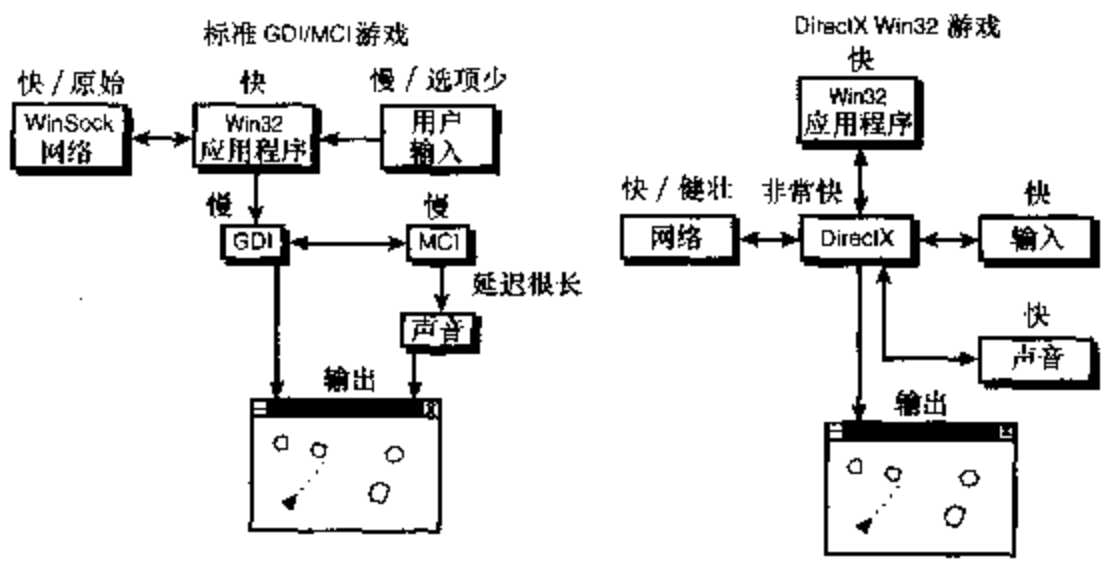


图 2.8 DirectX 与 GDI/MCI

那么 DirectX 是如何工作的呢？DirectX 几乎提供了对所有设备的硬件级控制。这是通过一种叫作 COM（Component Object Model，组件对象模型）以及微软公司和硬件供货商所编写的驱动程序集和相应库文件来实现的。微软公司提供了一套规则、函数、变量、数据结构等，硬件供货商在编写驱动程序时必须使用它们来与硬件进行对话。

只要编写驱动程序时遵循了这些规则，就无需考虑硬件细节，而只需调用 DirectX，后者将为您处理这些细节。不管使用哪种视频卡、声卡、网卡和其他硬件，只要它们支持 DirectX，程序就能够使用它们，而无需知道有关硬件的任何细节。

当前 DirectX 8.0+基本系统由很多 DirectX 组件组成，如图 2.9 所示：

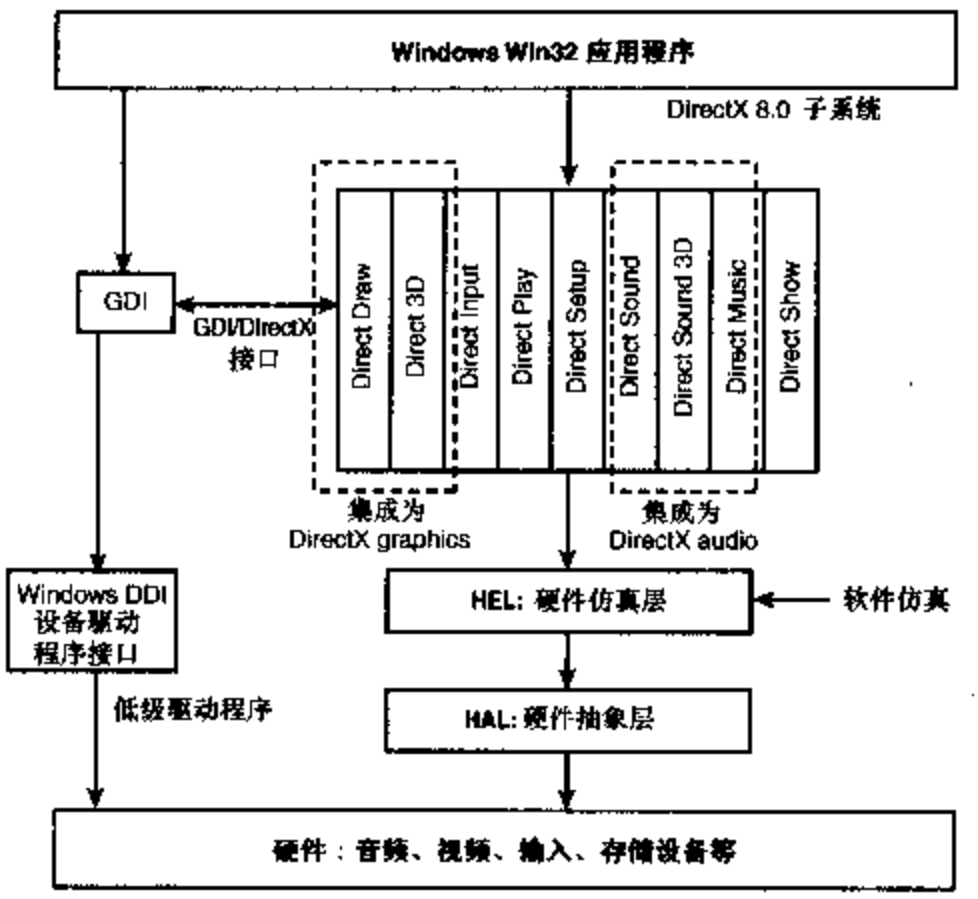


图 2.9　DirectX 8.0+的体系结构及其与 Win 32 的关系

- DirectDraw；
- DirectSound；
- DirectSound3D；
- DirectMusic；
- DirectInput；
- DirectPlay；
- DirectSetup；
- Direct3DRM；
- Direct3DIM。

在 DirectX 8.0 中，DirectDraw 与 Direct3D 结合为 DirectGraphics；DirectSound 与 DirectMusic 合并为 DirectAudio。因此，在 DirectX 8.0 以上版本中，不再有 DirectDraw。但是，DirectX 和 COM 的规则之一是，您总是可以请求以前版本的接口。因此，本书使用 DirectX 7.0a 中的 DirectDraw 进行图形处理，使用新的 8.0+接口进行声音和输入处理。我们将图形、声音和输入抽象到虚拟计算机中，所以甚至 DirectX 3.0 也管用！我们将使用最新版本的 DirectX（DirectDraw 除外），以保持尽可能最新。

2.4.1　HEL 和 HAL

读者可能注意到了，在图 2.9 中，DirectX 下面有两层，分别是 HEL（Hardware Emulation Layer，硬件仿真层）和 HAL（Hardware Abstraction Layer，硬件抽像层）。DirectX 是一种非常前卫的设计，它假定高级功能由硬件来实现。但是，如果硬件不支持某种功能，将发生什么事情呢？这就是双模式（HAL 和 HEL）设计的基础。

● HAL——硬件抽象层直接与硬件对话。这一层通常是来自硬件供货商的设备驱动程序，您通过常规 DirectX 调用直接与它通信。当硬件设备支持您请求的功能时，将使用 HAL，从而得到了硬件加速。

● HEL——当硬件设备不支持您请求的功能时，将使用硬件仿真层。假定您请求视频卡旋转一幅位图，如果硬件不支持旋转，将通过 HEL，使用软件算法来实现旋转。显然，使用 HEL 时的速度更慢，但它可以保持程序得以正常运行。它仍然能够工作——只是速度慢一些。此外，在 HAL 和 HEL 之间的切换对于您来说也是透明的。如果您请求 DirectX 做某项工作，并且由 HAL 直接完成，硬件将完成该工作；否则，将通过 HEL，调用软件仿真完成该工作。

您可能认为这里有很多软件层。这确实是一个问题，但是 DirectX 相当简洁，所以您使用它的惟一代价是一个或两个额外的函数调用。对于 2D/3D 图形、网络和声音加速来说，这个代价是相当小的。您可以想象去编写驱动程序来控制市场上所有的视频加速卡吗？它将可能需要花费上万年——或者只是一项不可能完成的工作。DirectX 实际上是微软公司和所有硬件供货商的一项巨大的分布式工程，为您提供了超高性能标准。

2.4.2 DirectX 基本类

现在来看一下每个 DirectX 组件及其功能：

● DirectDraw（只在 DirectX 7.0a 中可用，在 DirectX 8.0+中不可用）——它是主要渲染和 2D 位图引擎，用于控制视频显示。所有图形都必须通过显示，它可能是所有 DirectX 组件中最重要的一个。DirectDraw 对象代表系统中的视频卡。

● DirectSound——它是 DirectX 的声音组件。但是，它只支持数字声音，而不支持 MIDI。这个组件使您的工作更容易，因为您不再需要第三方声音系统授权就能完成您的声音处理。声音编程是一种巫术，在以前没有人愿意去为所有的声卡编写所有的驱动程序。因此，有几家厂商在市场上提供了声音库：Miles Sound System 和 DiamondWare Sound Toolkit。这两种系统功能都很强大，使得您可以简单地从 DOS 或 Win32 程序中直接加载并播放数字声音和 MIDI。但是，随着 DirectSound、DirectSound3D 和最新的 DirectMusic 等组件的引入，第三方声音库已经不再像过去那么有用了。

● DirectSound3D——它是 DirectSound 的 3D 声音组件。它使得您可以定位 3D 声音，并模拟 3D 声音效果。这种技术相对较新，但是迅速得以发展成熟。现在大多数声卡都支持硬件加速 3D 效果，包括多普勒变换、折射、反射等。但是，如果使用软件仿真，则不需要使用该组件。

● DirectMusic——DirectMusic 提供了对 DirectSound 不支持的 MIDI 技术的支持。DirectMusic 有一个新的 DLS 系统（Downloadable sounds，可下载声音），它使得您可以创建乐器的数字表示，并用 MIDI 控件播放它们。它非常类似于波表合成器，但是它是以软件方式实现的。而且，DirectMusic 具有一个新的人工智能 Performance Engine（性能引擎）系统，可以根据您提供的模板，实时地改变音乐。实质上，该系统可以动态地创建新的音乐。

● DirectAudio——在 DirectX 8.0+版本中，DirectSound 与 DirectMusic 被合并成一个统一的接口。当然，您仍然可以分别使用它们，也可以选择使用新的 DirectAudio 接口。

● DirectInput——该系统将处理所有输入设备，包括鼠标、键盘、游戏杆、轨迹球等。而且，DirectInput 现在支持强力反馈设备。这些设备具有电子机械激励器和力量传感器，使得您可以让用户感觉到游戏反馈的力量。

● DirectPlay——它是 DirectX 的网络组件，让您可以使用 Internet、调制解调器、直接连接或任何其他介质来建立抽象“连接”。DirectPlay 使得您可以建立这些连接而不需要知道任何关于网络的东西。您不需要编写驱动程序、使用套接字或类似的东西。另外，DirectPlay 还支持 session（正在进行中的游戏）和 lobby（玩家聚集的地方）的概念。而且，DirectPlay 不强制您使用任何多玩家网络体系结构——您想要做什么事情完全取决于您自己。DirectPlay 所做的全部事情就是为您发送和接收分组。它们所包含的内容以及它们是否可靠都由您自己决定。

● Direct3DRM（只在 DirectX 7.0a 中可用，在 DirectX 8.0+中不可用）——它是 Direct3D 保留模式（Retained Mode），是一个高级对象和基于帧的 3D 系统，您可用来创建基本的 3D 程序。它利用了 3D 加速功能，但速度不是最快。它对于简单的程序、3D 模型操作程序或极慢的演示程序是非常适用的。

● Direct3DIM——它是 Direct3D 直接模式（Immediate Mode），为 DirectX 提供了低级 3D 支持。最

开始时，它很难使用，且导致了与 OpenGL 的竞争。原来的直接模式使用执行缓存（execute buffers）——用于描述场景的数据和指令数组。但是，自 DirectX 5.0 之后，直接模式可以通过 DrawPrimitive() 和 DrawIndexedPrimitive() 函数，支持与 OpenGL 类似的接口。它让您能够将三角形条带、扇形等发送给渲染引擎，使用函数调用而不是执行缓存来修改状态。

- DirectGraphics——在 DirectX 8.0+中，不再支持 DirectDraw，但是从哲学角度来说，这并不完全正确。微软公司所做的是将所有图形软件合并到一个统一的接口 DirectGraphics 中，它将处理一切事情。它并没有任何 DirectDraw 接口，但您可以使用 Direct3D 接口执行 2D 功能，或者是使用 DirectDraw 7 和标准 DirectDraw。
- DirectSetup/AutoPlay——它们是伪 DirectX 组件，使得您可以在光盘放入到系统中后，从自己的应用程序安装 DirectX 到用户的计算机上，并直接启动游戏。DirectSetup 是一个函数集合，它们运行阶段 DirectX 文件加载到用户的计算机中，并在注册表中对它们注册。AutoPlay 是标准光盘子系统，它在光盘的根目录下查找 AUTOPLAY.INF 文件，如果发现该文件，则执行该文件中的批处理命令函数。

最后，您可能想知道为什么有这么多版本的 DirectX——它似乎每 6 个月更新一次。这一点在很大程度上是正确的。这是我们所处行业的特征——图形和游戏技术发展速度相当快。然而，由于 DirectX 基于 COM 技术，所以即使是您为 DirectX 3.0 编写的程序，也可以在 Direct 8.0 和 9.0 版本的 DirectX 上运行。

2.5　COM 简介

COM 在很多年之前，是作为关于一种新软件范例的简单白皮书被发明的，这种新软件类似于计算机芯片或 LEGO 积木块——您可以简单地把它们堆积到一起，然后它们将能正常工作。由于计算机芯片和 LEGO 积木块都知道如何成为计算机芯片和 LEGO 积木块，所以一切都将自动进行。要用软件实现这种技术，需要一些非常通用的接口，它们可能是各种类型的函数集。这就是 COM 所做的事情。

计算机芯片最重要的一个特点是，在设计中添加更多的计算机芯片时，不需要告诉其他芯片您已经改变了一些东西。然而，对于软件编程来说，这要稍微困难一些。您至少需要重新编译才能得到一个可执行文件。解决这个问题是 COM 的另一个目标。您应该能够对一个 COM 对象添加新功能，而不会影响使用原 COM 对象的软件。此外，COM 对象可以被改变而不需要重新编译原来的程序，这一特点是非常有用的。

由于可以升级 COM 对象而不需要重新编译原来的程序，这样您就可以升级自己的软件而不需要补丁程序和新版本的程序。例如，假设您有一个程序使用了 3 个 COM 对象：一个 COM 对象实现图形，另一个实现声音，还有一个用于实现网络功能，如图 2.10 所示。现在假设您已经卖出了 10 万份使用这些 COM 对象的程序，但是您不想发送 10 万份升级程序！为了更新图形 COM 对象，您只需要为用户提供新的图形 COM 对象，然后程序将自动使用新对象。非常简单，不需要进行重新编译或链接。当然，所有这些技术在底层是相当复杂的，并且编写自己的 COM 对象可能需要花费很多精力和时间，但是使用它们则相对简单。

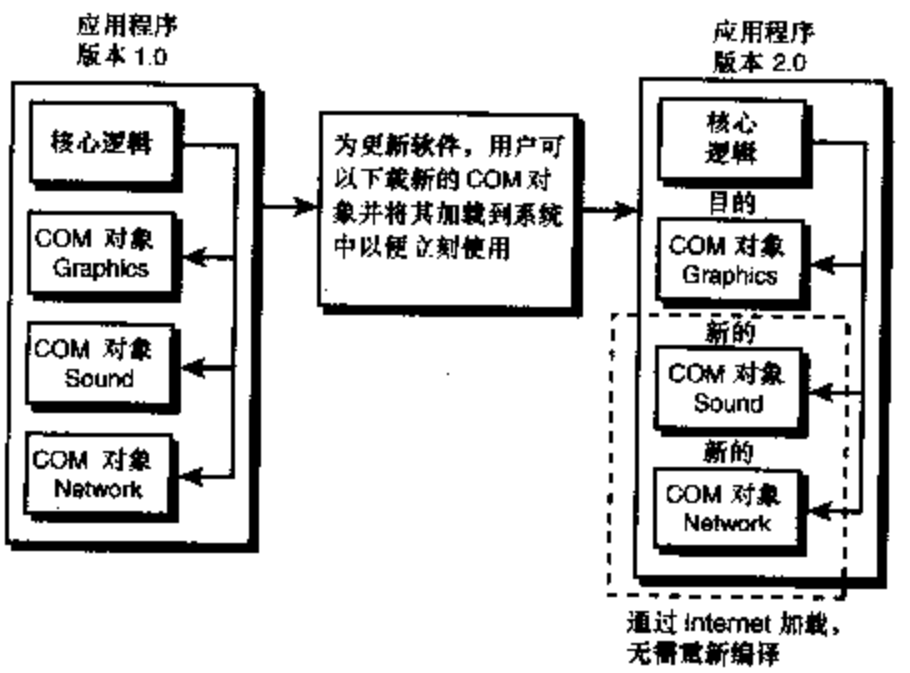


图 2.10　COM 概述

接下来的问题是如何分发 COM 对象？答案是关于这一点没有规则。但是，大多数情况下 COM 对象都是动态链接库（.DLL），可通过网络下载或随使用它们的程序一起提供。这样可以很容易地对它们进行修改和升级。惟一的问题是，使用 COM 对象的程序必须知道如何从.DLL 加载 COM 对象，后面将予以介绍。

2.5.1 什么是 COM 对象

COM 对象实际上是一个或一套 C++类，它实现了很多接口。这些接口用于与 COM 对象通信。实际上，接口就是执行某种服务的一套函数。如图 2.11 所示，我们看到了一个 COM 对象，它有 3 个接口，名称分别是 IGRAPHICS、ISOUND 和 IINPUT。

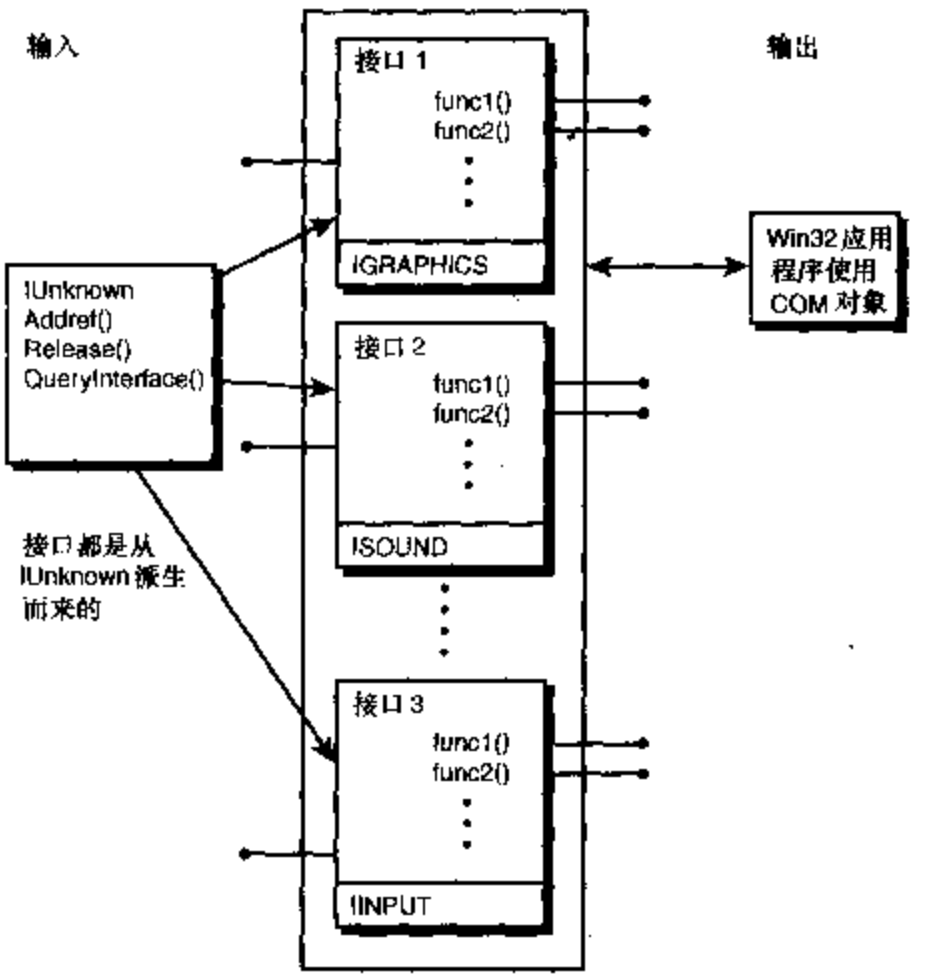


图 2.11 COM 对象的接口

每个接口都有很多函数，您可以调用这些函数来完成自己的工作。所以，一个 COM 对象可以有一个或多个接口，而您可以有一个或多个 COM 对象。COM 规范规定您创建的所有接口都必须从一个名为 IUnkown 的特殊基类接口派生得到。对于 C 程序员来说，这就意味着 IUnkown 是用于建立接口的起点。

下面来看一下 IUnkown 的类定义：

```
struct Iunknown
{

// 这个函数用于检索其他接口
virtual HRESULT __stdcall QueryInterface(const IID &iid, (void **)ip) = 0;

// 这个函数用于将接口引用数加 1
virtual ULONG __stdcall AddRef() = 0;

// 这个函数用于将接口引用数减 1
virtual ULONG __stdcall Release() = 0;

};
```

注意：所有方法都是纯粹（pure）和虚拟的。此外，这些方法使用遵循标准 C/C++调用规则的__stdcall。记住，__stdcall 按照从右至左的次序将参数放入堆栈。

即使您是一名 C++程序员，如果对虚函数不是特别熟悉，对于上面的类定义也会感到有点奇怪。不管怎样，让我们来分析一下 IUnkown。所有从 Iunkown 派生而来的接口都必须至少实现 3 种方法：

QueryInterface()、AddRef()和 Release()。

QueryInterface()是 COM 的关键——它用于请求一个指向所需接口函数的指针。要执行这种请求，必须提供一个接口 ID。该接口 ID 是您分配给接口的惟一数字，长度为 128 位，因此总共可以有 2^{128} 个不同的接口 ID，这在未来相当长的时间内都足够用。本章后面将会介绍一个与 ID 有关的真实范例。另外，COM 的规则之一是，如果有一个接口，总是可以在该接口中请求相同 COM 对象中的任何其他接口。图 2.12 说明了这一点。

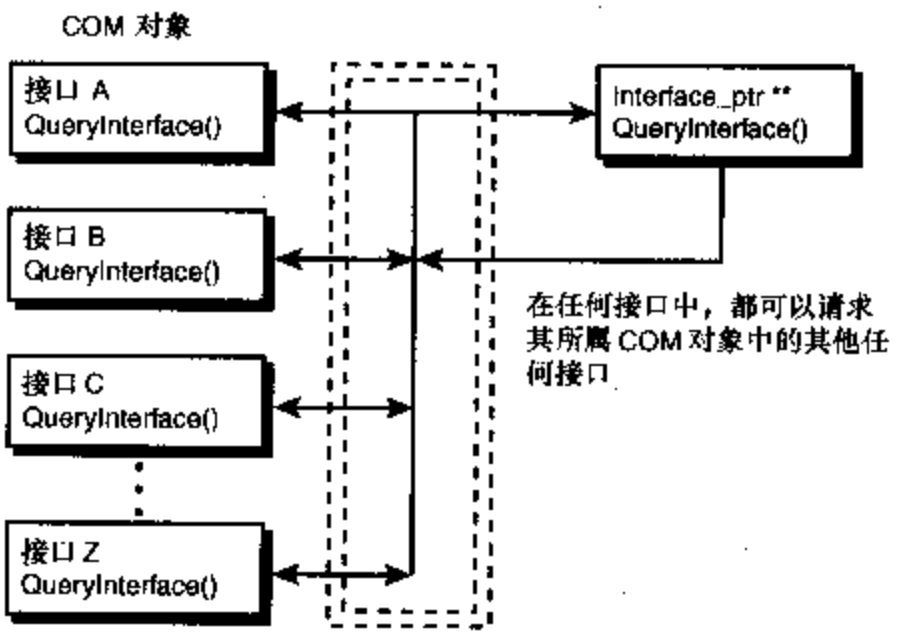


图 2.12　在 COM 对象的接口间导航

AddRef()是一个奇怪的函数。COM 对象使用一种叫作“引用计数”（reference counting）的技术来跟踪它们的生存期。这是因为 COM 的规范之一是，它不是语言特定相关的。因此，创建一个 COM 对象并创建接口以跟踪指向该对象的引用数量时，将调用 AddRef()函数。如果 COM 对象使用 malloc()或 new[ ]函数，它将成为 C/C++语言特定相关的。当引用计数降为 0 时，则内部销毁该对象。

提示：通常您不需要自己对接口或 COM 对象调用 AddRef( )函数；它将由 QueryInterface( )函数内部完成。但是，有时您可能想要增加引用计数，让 COM 对象以为有比实际值更多的引用数量，在这种情况下需要自己调用该函数。

这提出了一个问题：如果 COM 对象是 C++类，如何使用 Visual Basic、Java、Active X 等来创建并使用 COM 对象呢？COM 的设计者使用 Visual C++类来实现 COM 对象，但是在使用或创建 COM 对象时并不一定需要使用 C++语言。只要您创建与 Microsoft C++编译器相同的二进制镜像，该 COM 对象就是 COM 兼容的。当然，大多数编译器都附带有额外的工具，用于帮助创建 COM 对象，因此这并不是一个很大的问题。您可以用 C++、Visual Basic 或 Delphi 来编写 COM 对象，然后用任何语言来访问该对象！内存中的二进制镜像仅仅是内存中的二进制镜像。

Release()函数用于将 COM 对象或接口的引用计数减 1。在大多数情况下，当您完成对一个接口的操作后，需要手工调用该函数。但是，有时候您创建一个对象，然后从该对象创建另一个对象，则对父对象调用 Release()函数将导致对子对象或派生对象调用 Release()函数。但是不管怎样，最好按相反的次序来调用 Release()函数。

2.5.2　创建和使用 DirectX COM 接口

我们知道 COM 对象是一些接口的集合，这些接口是一些函数指针，具体地说是 VTABLE。因此，对于 DirectX COM 对象，您所需要做的全部工作就是创建它，获取一个接口指针，然后使用适当的语法调用该接口。当然，这只是为了便于您理解，我们将使用 API 来完成这些工作。我想您可能根本不会用到 COM 调用，但最好还是知道这些。我将以主 DirectDraw 接口为例来演示如何完成这些工作。

首先，要实验使用 DirectDraw，需要完成 3 项工作：

- 必须被加载并注册 DirectDraw 运行阶段 COM 对象和.DLL。这项工作是由 DirectX 安装程序完成的。
- 必须在 Win32 程序中包含 DDRAW.LIB 导入库，以便将调用的包装函数链接到程序中。
- 需要在程序中包含 DDRAW.H 文件，让编译器能够“看到”DirectDraw 的头信息、函数原型和数据类型。

下面是 DirectDraw 1.0 接口指针的数据类型：

```
LPDIRECTDRAW lpdd = NULL;
```

要创建一个 DirectDraw 1.0 COM 对象并获取一个指向 DirectDraw 对象（它代表视频卡）的接口指针，只需使用包装函数 DirectDrawCreate()：

```
DirectDrawCreate(NULL, &lpdd, NULL);
```

不必关心其中的参数。如果您有兴趣，可以查看 DirectX SDK，但是 99%的时候该函数的样子将与上面相同。也就是说，它只使用一个参数：将用 DirectDraw COM 接口填充的接口指针地址。就现在而言，您只需知道该调用创建一个 DirectDraw 对象，并将接口指针赋给 lpdd 即可。

注意：当然，该函数实际上执行了很多操作。它打开一个.DLL，执行函数调用并完成很多其他工作，但您不用关心其具体细节。

现在一切准备就绪，可以调用 DirectDraw 了。但是稍等一下！我们不知道有哪些方法和函数可用——这也是您为什么阅读本书的原因。下面的代码将视频模式设置为 8 位（256）彩色、640×480 模式：

```
lpdd->SetVideoMode(640, 480, 8);
```

是不是很简单？惟一额外的工作是对 DirectDraw 接口指针 lpdd 解除引用。当然，实际上需要查询虚拟接口表，但我们不用关心这些。

实质上，任何对 DirectX 的调用都采用如下形式：

```
interface_pointer->method_name(parameter list);
```

另外，对于要使用的任何其他接口（如 Direct3D），需要在原始的 DirectDraw 接口中使用 QueryInterface() 函数来获得。由于 DirectX 的版本有很多，所以微软公司不再编写用于获取最新接口的包装函数。这意味着您必须使用 QueryInterface() 函数来手工获取最新的 DirectX 接口。下面介绍这方面的内容。

2.5.3 查询接口

关于 DirectX 最奇怪的事情是所有版本号的不同步。有时候这将成为问题，并可能引起混乱。第一个版本的 DirectX 出现时，DirectDraw 接口的命名方式与下面类似：

```
IDIRECTDRAW
```

然后当 DirectX 2.0 出现后，DirectDraw 也升级到版本 2.0，因此有了：

```
IDIRECTDRAW

IDIRECTDRAW2
```

到版本 7.0a 时，有了：

```
IDIRECTDRAW

IDIRECTDRAW2

IDIRECTDRAW4

IDIRECTDRAW7
```

在版本 8.0 和版本 9.0 中，不再有 IDIRECTDRAW 升级，因为它被合并到 Direct3D 中，但其他接口仍然可用：

```
IDIRECTDRAW
```

```
IDIRECTDRAW2

IDIRECTDRAW4

IDIRECTDRAW7
```

我将 8.0 和 9.0 版本相提并论，是因为这两个版本之间没有什么大的区别。在 9.0 版本中惟一的区别是，它提供了更高级的 3D 支持、更多的 DirectPlay 软件和其他一些改进。重要的修改发生在 7.0 版本到 8.0 版本的升级中。读者可能会问，接口 3 和 5 之间有什么区别呢？我不知道，这也正是问题所在。即使您使用的是 DirectX 8.0+，但并不意味着所有接口都升级到了该版本；另外，它们可能是不同步的。在 DirectX 6.0 中，DirectDraw 接口的最高版本是 IDIRECTDRAW4，但 DirectSound 的最高版本为 1.0（被称为 IDIRECTSOUND）。正如您看到的，这些是多么混乱！介绍这些内容旨在说明一个原则，无论何时使用 DirectX 接口，务必使用最新版本的接口。如果您不能肯定，应使用通用“Create”函数来获得最新的版本。下面是一个范例。

DirectDrawCreate()返回修改版 1.0 接口指针，但 DirectDraw 的最高版本为 IDIRECTDRAW7，我们该应该如何利用这项新功能呢？

有两种方法可以根据 DirectDraw 1.0 接口实现该目标：使用低级 COM 函数或使用 QueryInterface()函数；我们采用后一种方法，步骤如下：

1. 调用函数 DirectDrawCreate()来创建 DirectDraw COM 接口，它将返回一个 IDIRECTDRAW 接口指针。

2. 使用步骤 1 获得的指针，用 IDIRECTDRAW7 的接口 ID（GUID）作为参数来调用 QueryInterface()函数，这样便可获得新接口。

下面是一个查询 DirectDraw7 接口的范例：

```
LPDIRECTDRAW lpdd;  // 1.0版
LPDIRECTDRAW7 lpdd7; // 7.0版

// 创建1.0版DirectDraw对象接口
DirectDrawCreate(NULL, &lpdd, NULL);

// 在头文件DDRAW.H中查找IDIRECTDRAW7接口的ID
// 然后使用它来查询该接口
lpdd->QueryInterface(IID_IDirectDraw7, &lpdd7);
```

这时候，您有了两个接口指针，但您不再需要指向 IDIRECTDRAW 的指针，因此应释放该指针：

```
// 释放指针，引用计数减1
lpdd->Release();

// 出于安全考虑，将其设置为NULL
lpdd = NULL;
```

别忘了，使用完任何接口后都应将其释放，因此程序结束时，需要释放 IDRECTDRAW7 接口：

```
// 释放接口，引用计数减1
lpdd7->Release();

// 出于安全考虑，将其设置为NULL
lpdd7 = NULL;
```

您可能会问，是否有用于获取 DirectDraw 7.0 接口的辅助函数呢？的确有这样的函数，但为了演示 QueryInterface()函数，我直到现在才指出这一点，请读者不要责怪。下面是更容易的 DirectDraw 7.0 接口获取方法：

```
LPDIRECTDRAW7 lpdd7; // 7.0版接口

// 请求接口
DirectDrawCreateEx(NULL, (void **)&lpdd, IID_IDirectDraw7, NULL);
```

该函数与 DirectDraw 1.0 版几乎相同，但参数稍有不同，如下所示：

```
HRESULT WINAPI DirectDrawCreateEx(
 GUID FAR *lpGUID, // 驱动程序的 GUID，对于活动显示器（active display）为 NULL
 LPVOID *lplpDD,  // 接口的接收方
 REFIID iid,    // 请求的接口的 ID
 IUnknown FAR *pUnkOuter // 高级 COM，NULL
);
```

有关如何使用 DirectX 和 COM 就介绍到这里。当然，这里并没有介绍 DirectX 组件的所有函数（有数百个）和所有接口。

2.6 总　　结

至此，您掌握了一些基本的 Windows 编程知识、什么是 DirectX 及其与 Windows 的关系。读者无需知道太多有关 DirectX 或 Windows 的知识，就能理解本书的大部分内容，因为下一章将建立一个虚拟计算机，用于运行我们编写的 3D 试验程序。这样，读者将无需了解 Wind32/DirectX 的底层机制，从而可以将主要精力放在 3D 图形学上。有关 Win32/DirectX 的知识很有用，要更深入地了解这方面的信息，请参阅《Windows 游戏编程大师技巧》或其他优秀的 Windows 编程图书。

第 3 章　使用虚拟计算机进行 3D 游戏编程

本章将建立一个基于软件接口的虚拟计算机系统，它将支持线性 8 位/16 位帧缓存（双缓存）、输入设备以及声音和音乐处理功能。有了这个接口，本书剩余部分便可以将重点放在 3D 数学、图形和游戏编程上。本章包括以下内容：

- 设计一个虚拟计算机图形接口；
- 建立一个 Windows 游戏控制台；
- 《Windows 游戏编程大师技巧》中增强 T3DLIB 库的 API 清单；
- 使用 T3DLIB 实现虚拟计算机；
- 最终的游戏控制台；
- 使用 T3DLIB 游戏库。

3.1　虚拟计算机接口简介

本书的目标是介绍 3D 图形学和游戏编程，但作者面临的困境是，如何将重点放在上述主题上，而不过多地介绍有关 Win32 编程、DirectX 等的低级细节。尽管阅读前一章后，您对 Win32 编程和 DirectX 有了一定的了解，但那只是入门介绍。我的解决方案是以《Windows 游戏编程大师技巧》中开发的引擎为基础，创建一台“虚拟计算机”，您可以将它作为黑盒使用，这样便可以将主要精力放在 3D 图形和游戏编程上。然而，由于该引擎非常大，因此仍有很多内容需要讨论。

注意：这基本上也是 OpenGL 采用的方法。您使用 GL 库和附件来处理低级设备特定接口和清理工作，包括打开窗口和获得输入。

基于众多的原因，这个引擎是很有意义的。只要能够与一个支持输入、声音和音乐的双缓存、线性寻址图形系统通信，谁会在乎它的工作原理呢？我们的兴趣在于 3D 图形编程，而不是低级设置。由于我们 99%的工作都与光栅化、纹理映射、光照、隐藏面消除等相关，所以这种方法是可行的。因此，我们需要的是一个黑屏，与游戏杆、键盘和鼠标交互的能力，有时候还需要播放一些声音效果和音乐。

因此，本章不试图去解释 Win32、DirectX 基础等细节，而是《Windows 游戏编程大师技巧》中编写的 API，来创建一台通用虚拟计算机，再在其上进行 3D 图形试验和编写游戏。

有了这个间接层，大多数 3D 代码的都足够通用，可以很容易地移植到 Mac 或 Linux 平台上。您需要做的是模拟低级接口，如双缓存图形系统、声音、音乐和输入设备。对于所有 3D 代码，算法仍然是相同的。

使用《Windows 游戏编程大师技巧》中引擎的惟一缺点是，有些数据结构和设计本身都是以 DirectX 为基础的。出于性能方面的考虑，我希望到 DirectX 的这一层最薄。因此，本可以在这一层上编写另一个完全与机器无关的软件层，但我认为这样做不值得。我的最低底线是，要移植 3D 内容，只要能够创建双缓存显

示，一切都将正常工作。您可以完全不管声音、音乐和输入，因为引擎中的函数只是调用了 DirectX 函数。

然而请牢记，只要理解了我提供的 API，将不需要了解有关 Win32 和 DirectX 的任何知识，因为我们对 3D 图形所做的任何处理都是由这些 API、帧缓存和 C/C++代码组成的。

本书最大的一个特点是，只关注 3D 图形和游戏编程。我们不用考虑所有关于设置 DirectX、获得输入和生成声音等的细节，而只是调用 API 函数。为此，我们将制定虚拟（抽象）图形计算机规范，然后使用《Windows 游戏编程大师技巧》中的 API 函数实现它。虚拟计算机的主旨在于帮助我们将精力集中在 3D 而不是设置细节上。下面是在世界上任何一台计算机上编写 3D 游戏所需的功能：

- 能够使用所需的位深在显示器上创建窗口，窗口是一个可线性寻址的 2D 像素阵列。另外，还需要支持离屏页（双缓存），以便能够离屏渲染图像，然后将其复制到主显示器中，以生成流畅的动画。
- 能够从系统输入设备（如键盘、鼠标、游戏杆等）获得输入。这些输入应该是原始输入，且采用简单格式。
- （可选）能够加载并播放常见格式（如.WAV 和.MID）的音乐。

图 3.1 是我们要用软件设计的虚拟计算机的系统级示意图。我过去常说，“只要能够绘制像素和读取键盘输入，就能够编写 Doom。”的确是这样，您只需能够访问帧缓存，其他一切都很简单。由于本书主要介绍软件 3D 编程和光栅化，因此所有算法都没有使用任何形式的 3D 加速。我们将绘制多边形、进行光照计算、对帧缓存中的每个像素进行其他处理。

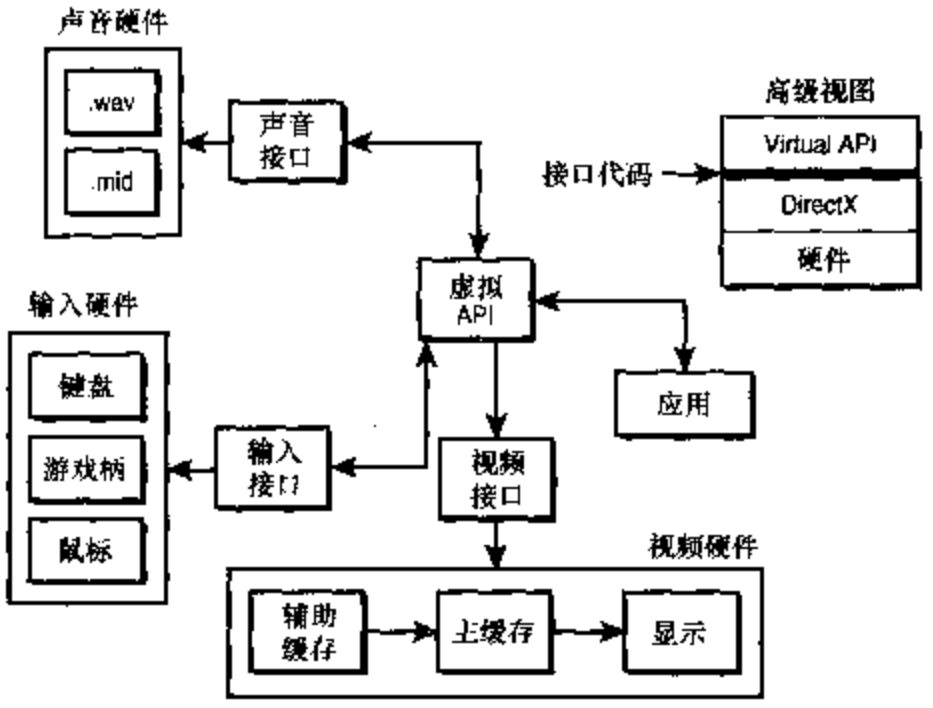


图 3.1 虚拟计算机的系统级示意图

注意：您可能会说，在可以使用硬件的情况下，为什么使用软件呢？原因有 3 个：首先，要成为优秀的图形编程人员，必须知道如何自动动手来完成各项任务；其次，了解工作原理有助于理解硬件加速；最后，谁来编写硬件代码呢？

3.2 建立虚拟计算机接口

有了能够实现前一节要求的功能的函数库后，建立虚拟计算机接口将相当容易。当然，我们有这样的函数库，就是《Windows 游戏编程大师技巧》中创建的函数库。介绍具体细节之前，先来设计虚拟计算机接口所需的函数和数据结构“模型”。当然，这只是一个范例，是非常高级的。另外，我将在这个过程中给出函数名称，以方便引用。虚拟计算机的最终版本将与此不同，因为在实现它时必须考虑细节。然而，下面的练习将让您对虚拟计算机有大致的了解。

3.2.1 帧缓存和视频系统

开始之前，假设有一个方法可以初始化视频系统，并以某种分辨率和位深打开窗口。我们称之为

Create_Window()，其原型如下：

```
Create_Window(int width, int height, int bit_depth);
```

要以 8 位颜色模式创建一个 640×480 的窗口，可以这样调用上述函数：

```
Create_Window(640, 480, 8);
```

要以 16 位颜色模式创建一个 800×600 的窗口，可以这样调用它：

```
Create_Window(800, 600, 16);
```

这只是一个学术模型，实际创建窗口和初始化系统的代码可能是一组函数。实际上，“窗口”可能是 Windows 窗口，也可能是全屏，所以创建窗口不会这么简单。现在假设已经有了一个窗口。

正如前面指出的，我们将设计一个这样的系统，即基于一个可见的主显示缓存和一个不可见的离屏辅显示缓存。这两个缓存都应该是可线性寻址的，其中每个字（word）代表一个像素，根据位深的不同，字可能是 BYTE、WORD 或 QUAD。图 3.2 描述了帧缓存系统。

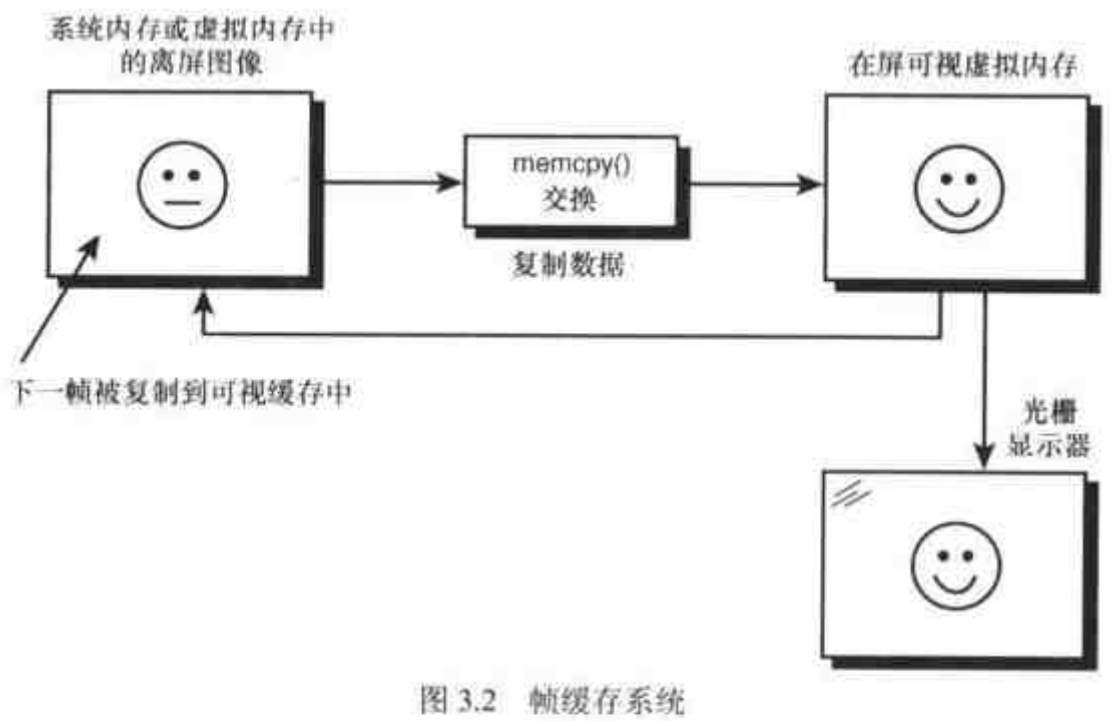


图 3.2　帧缓存系统

在主显示缓存和辅显示缓存各自的帧缓存内存中，都有一个“内存填充”（memory padding）区域。这是因为在有些视频系统中，每行的像素是连续存储的，但行与行之间不是连续的，图 3.3 清晰地说明了这一点。在该范例中，视频系统被设置为 640×480×8，因此每个像素应为 1 字节，每个视频行应为 640 字节，

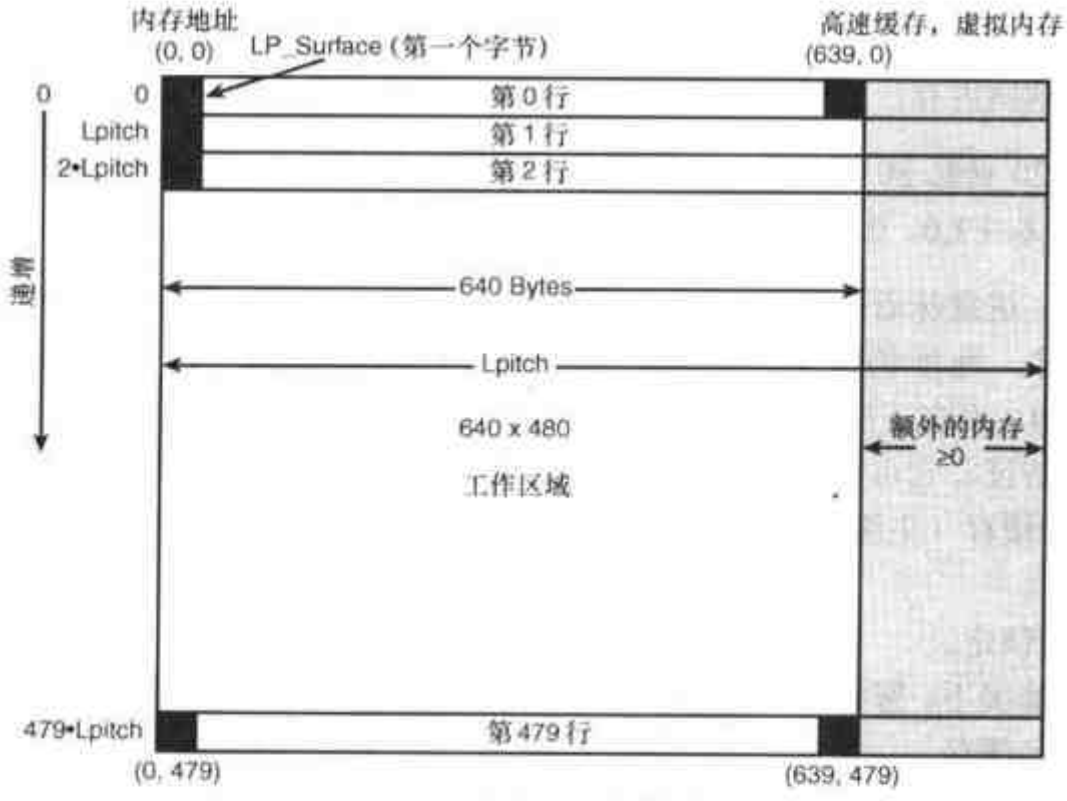


图 3.3　硬件帧缓存详图

但实际上并非如此，每个视频行为 1024 字节。这是视频卡内存寻址的一种特征，很多视频卡都这样做。因此，需要使用一种通常被称为内存跨距（memory pitch）的机制来处理这种不一致性。

内存跨距反映了视频卡可能由于高速缓存或硬件寻址的原因，每行有额外内存的事实。只要不假定内存跨距等于视频跨距（video pitch，这里为 640 字节），就不会导致问题。在每像素 8 位的视频系统中，要确定像素的地址，可以使用下述代码：

```
UCHAR *video_buffer; // 指向视频缓存的指针
int x,y;       // 像素的坐标
int memory_pitch;  // 每行多少字节

video_buffer[x + y*memory_pitch] = pixel;
```

在 16 位系统中，使用稍微不同的代码：

```
USHORT *video_buffer; // 指向视频缓存的指针
int x,y;       // 像素的坐标
int memory_pitch;   // 每行多少字节

video_buffer[x + y*(memory_pitch >> 1)] = pixel;
```

当然，如果 memory_pitch 就是每行的 USHORT 数，则无需使用移位运算符>>将其除以 2。

下面来创建主缓存和辅助缓存的模型。别忘了，这只是一个模型，使用 T3DLIB 游戏引擎实现该模型时，细节将会有所不同。我们需要指向主缓存和辅助缓存的指针以及用于存储内存跨距的变量。例如，下面是一些用于上述目的的全局变量：

```
UCHAR *primary_buffer; // 主缓存
int  primary_pitch;  // 内存跨距，单位为字节

UCHAR *secondary_buffer; // 辅助缓存
int  seconday_pitch;   // 内存跨距，单位为字节
```

这两个指针都是 UCHAR*类型，且内存跨距以字节为单位。使用 8 位颜色模式时，这很好，但如果使用 16 位颜色模式呢？在这种模式下，每个像素 2 字节，字为 USHORT。这样做的原因在于，我想保持数据类型尽可能简单、一致。如果使用的是 16 位模式，只需将指针强制转换为 USHORT*。将指针设置为 UCHAR*更为清晰，且我们知道内存跨距总是以字节为单位。这样，我们编写的所有函数都有相同类型的指针/跨距参数。

锁定内存

还需要另一项功能——锁定内存。很多视频卡都有特殊内存，可能是多端口（multi-ported）内存、高速缓存等。这意味着当您访问这些内存和读写帧缓存时，需要让系统知道您正在访问这些内存，这样系统在此期间将不会修改这些内存。完成读写后，需要解除对这些内存的锁定，以便系统可以处理其他业务。

提示：如果您是 20 世纪 80 年代或 90 年代初的 Windows 编程大师，对内存锁定应当非常熟悉，因为在早期的 Windows 版本（1.0、2.0 和 3.0）中，必须锁定内存。

对于程序员来说，这意味着指针 primary_buffer 和 secondary_buffer 只在锁定期间有效。另外，您不能认为在下一次锁定时，地址仍然是相同的。也就是说，可能在第一次锁定-解锁期间，主缓存位于 0x0FFFEFFC00000000，而在下次锁定期间则位于 0x0FFFFFFD00000000。

这是硬件的一种特征。它可能会移动帧缓存，因此一定要小心！内存锁定步骤如下：

1. 锁定感兴趣的缓存（主缓存或辅助缓存），并获取起始内存地址和内存跨距。
2. 操作视频内存。
3. 解除对缓存的锁定。

当然，在 99%的情况下，您将对辅助缓存而不是主缓存执行锁定-读写-解锁操作，因为您不想用户“看到”您正在修改主显示缓存。

下面来创建几个执行锁定和解锁操作的函数：

```
Lock_Primary(UCHAR **primary_buffer, int *primary_pitch);
Unlock_Primary(UCHAR *primary_buffer);

Lock_Secondary(UCHAR **secondary_buffer, int *secondary_pitch);
Unlock_Secondary(UCHAR *secondary buffer);
```

为锁定缓存，您使用相应帧缓存的地址和内存跨距作为参数调用上述函数。该函数将锁定缓存，并修改（写）传入的变量。然后就可以随便使用它们。完成操作后，需要使用一个指向锁定缓存的指针作为参数，调用 Unlock_*() 函数，以解除对缓存的锁定。

现在来看看在 8 位模式和 16 位模式下，如何将一个像素写入到分辨率为 800×600 的屏幕的辅助缓存中。下面是一个 8 位模式范例：

```
UCHAR *primary_buffer; // 主缓存
int  primary_pitch; // 内存跨距，单位为字节

UCHAR *secondary_buffer; // 辅助缓存
int  seconday_pitch;   // 内存跨距，单位为字节

UCHAR pixel; // 要写入的像素
int x,y;   // 像素的坐标

// 第 0 步：创建窗口
Create_Window(800, 600, 8);

// 第 1 步：锁定辅助缓存
Lock_Secondary(&secondary_buffer, &secondary_pitch);

// 将像素写入到坐标指定的位置
secondary_buffer[x + y*secondary_pitch] = pixel;

// 完成写入操作后，解除对辅助缓存的锁定
Unlock_Secondary(secondary_buffer);
```

这很容易！下面系统是处于 16 位颜色模式下（每个像素 2 字节）的范例：

```
UCHAR *primary_buffer; // 主缓存
int  primary_pitch;  // 内存跨距，单位为字节

UCHAR *secondary_buffer; // 辅助缓存
int  seconday_pitch;   // 内存跨距，单位为字节

USHORT pixel; // 要写入的 16 位像素
int x,y;   // 像素的坐标

// 第 0 步：创建窗口
Create_Window(800, 600, 16);

// 第 1 步：锁定辅助缓存
Lock_Secondary(&secondary_buffer, &secondary_pitch);

// 由于被锁定的指针为 UCHAR *，而不是 USHORT *
// 因此需要进行强制类型转换

USHORT *video_buffer = (USHORT *)secondary_buffer;

// 根据像素的坐标，将像素写入到屏幕中相应的位置
video_buffer[x + y*(secondary_pitch >> 1)] = pixel;

// 执行完写入操作后，解除对辅助缓存的锁定
Unlock_Secondary(secondary_buffer);
```

由于内存跨距是以字节为单位的，为正确地使用内嵌的指针算术运算，需要将其除以 2（使用运算符 >>），将内存跨距转换为每行的 USHORT 数。

最后，还有一个细节没有介绍，这就是 8 位和 16 位像素数据格式。

3.2.2 使用颜色

出于速度方面的考虑，我们将在 3D 游戏编程中主要使用 8 位调色板视频模式和 16 位 RGB 模式。在 8 位颜色模式下，每个像素为 1 字节；在 16 位颜色模式下，每个像素为 2 字节。然而，它们的编码方法是完全不同的。

1. 8 位颜色模式

8 位颜色模式使用标准的 256 项颜色查找表（color lookup table，CLUT），如图 3.4 所示；而 16 位颜色模式使用标准的 RGB 编码方案。使用 8 位颜色模式时，必须用 RGB 值填充颜色查找表中的 256 个颜色项。我们假设有一个用于访问这个表和读取/修改每个颜色项的函数，因此现在暂时不去考虑其细节。

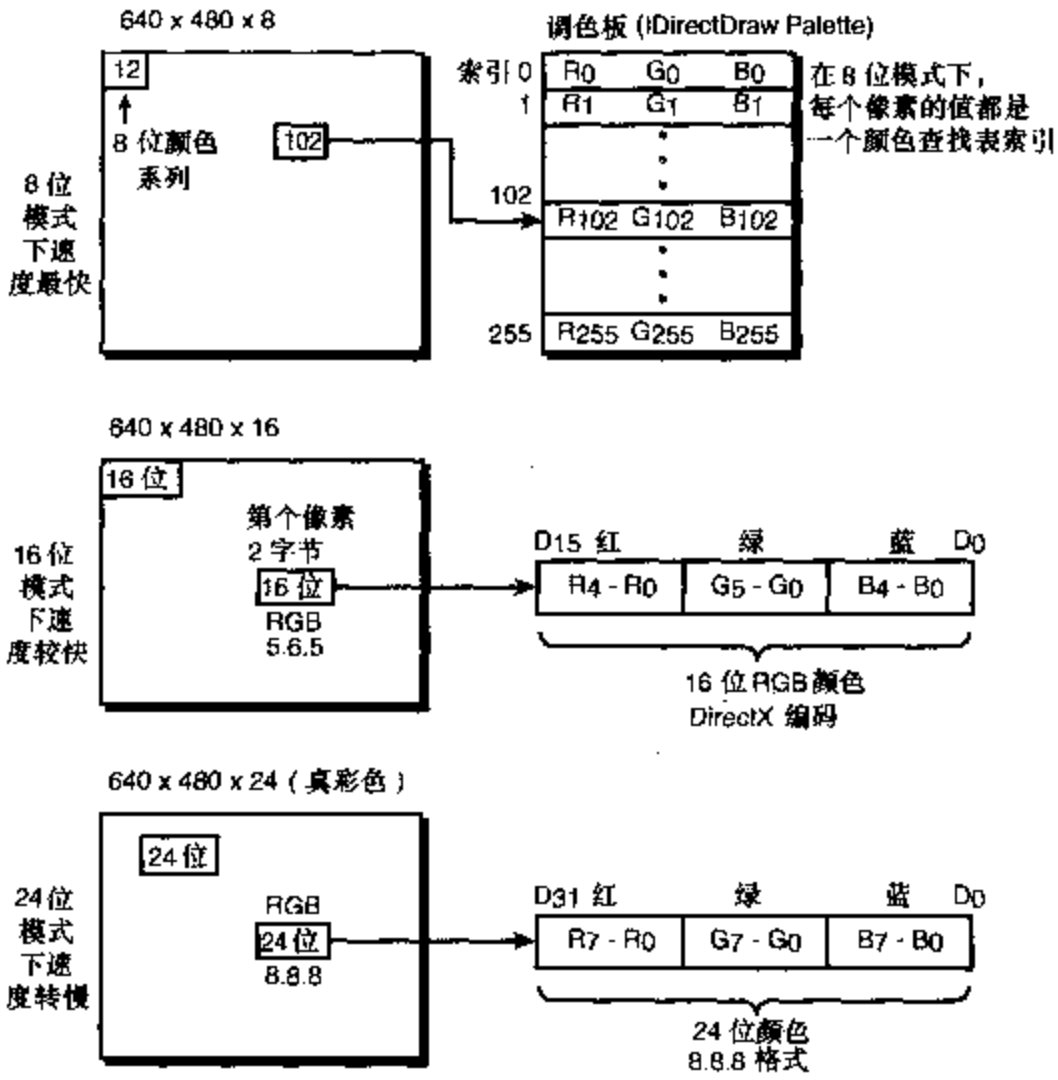


图 3.4　不同位深及其实现的比较

8 位颜色模式没有什么特别的，它是标准的每像素 1 字节系统，每个像素的值都是一个颜色查找表索引，而每个表项都代表一种颜色，该颜色由 3 个 8 位的颜色分量（红、绿、兰）组成。

提示：有些显示卡只使用 8 位颜色分量中的 6 位，也就是说，对于每种原色（红、绿、兰）只有 $2^6 = 64$ 种着色度（shade），而不是标准的 $2^8 = 256$ 种着色度。

2. 16 位颜色模式

使用 16 位颜色要比使用 8 位颜色简单些。在 16 位颜色模式下，不需要颜色查找表。实际上，每个像素是由一系列位组成的 RGB 字。在这种模式下，像素为 5.5.5（RGB）或 5.6.5（RGB）格式：也就是说，5 位红色、5 位绿色、5 位蓝色，或者是 5 位红色、6 位绿色、5 位蓝色，如图 3.5 所示。这种 RGB 像素格式有时会令人困惑，我就收到了很多提出这方面问题的电子邮件。常见的问题是如何生成像素？我认为问题在于，每个人都把它想得比实际情况复杂，它实际上只是一些位操作。

在 5.5.5 RGB 模式下，每个分量为 5 位，即每个颜色分量有 $2^5=32$ 种着色度。因此，R、G、B 的取值范围为 0～31，最多有 32*32*32＝32 768 种颜色。因此，您需要做的就是编写一些代码，处理这 3 个值。假设有 3 个变量 r、g、b，取值范围都是 0～31（31 为最大颜色亮度），可以使用位运算符来生成像素。例如，对于 5.5.5 模式，可以使用下面的宏：

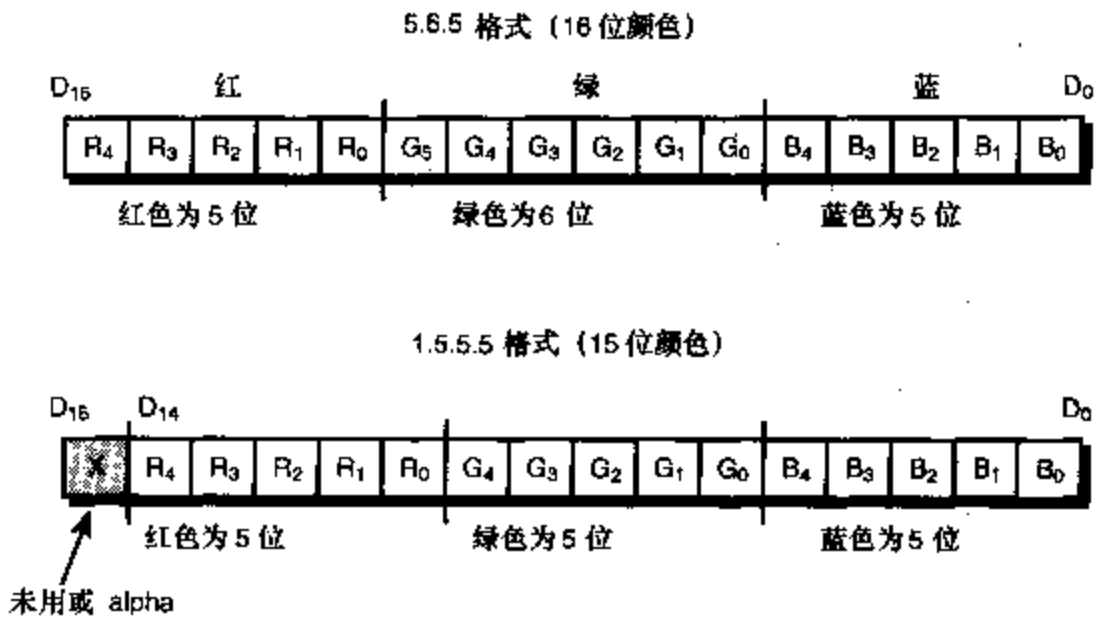


图 3.5　5.5.5 格式和 5.6.5 格式的 16 位颜色编码方案

```
// 生成一个 5.5.5 格式的 16 位颜色值(alpha 占 1 位)
#define _RGB16BIT555(r,g,b) ((b & 31) + ((g & 31) << 5) + ((r & 31) << 10))
```

同样，在 5.6.5 模式下，绿色分量为 6 位，因此取值范围如下：红色为 0～31，绿色为 0～63，蓝色为 0～31。这可以提供 32*64*32＝65 536 种颜色。同样，可以使用下面的宏根据 r、g、b 值生成一个像素：

```
// 生成 5.6.5 格式的 16 位颜色值（绿色分量为 6 位）
#define _RGB16BIT565(r,g,b) ((b & 31) + ((g & 63) << 5) + ((r & 31) << 11))
```

这个问题在过去给人们带来了很多麻烦，我们将在系统中采取一些方法来处理它。图形系统在被选择进入 16 位模式时，将自动确定像素格式（5.5.5 模式或 5.6.5 模式）并设置标记和函数指针，这样您可以使用相同的宏来生成 16 位颜色的像素。这项功能将出现在后面建立的“真实”库中。下面介绍缓存交换。

3.2.3　缓存交换

需要实现的最后一项功能是缓存交换，即将辅助缓存复制到主缓存中，如图 3.6 所示。当然，可以同时锁定两个缓存，然后逐行进行复制。但是，很多显卡都有特殊的硬件，用于以很快的速度进行交换和复制，我们将支持后一种方法，使用黑盒方法通过 API 来实现缓存交换。我们将该函数命名为 Flip_Display()，它不接受参数，将辅助缓存复制到主缓存（使用软件方法还是硬件方法，我们并不知道）。

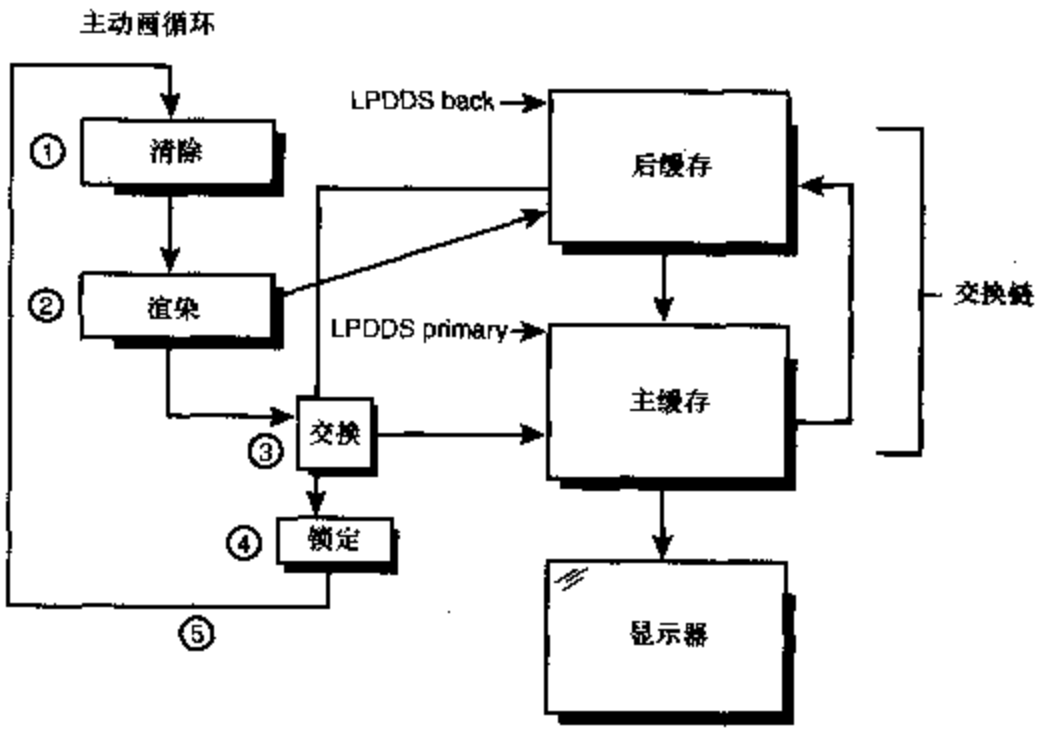


图 3.6　页面交换动画系统

警告：您可能会问，为何要逐行复制辅助缓存，而不使用 memcpy( )函数。这是因为辅助缓存的行与行之间可能是连续的，也可能是不连续的，而 memcpy( )函数没有办法知道这一点。所以，必须根据内存

跨距逐行复制缓存，然后处理每一行。

然而，我们将遵循一条规则：调用 Flip_Display() 函数时，必须确保主缓存和辅助缓存都没有被锁定；否则，将不会执行交换，且会引发错误，甚至可能导致计算机死机。因此，调用 Flip_Display() 函数前，应确保两个缓存都没有被锁定，系统能够访问它们。

另外，生成动画时，有两个非常方便的辅助函数：用于清除缓存的函数。同样，也可以锁定缓存，然后使用 memset() 或其他函数来逐行清除缓存，但很多显示卡有用于清除缓存的硬件填充功能。我们假设显卡支持硬件填充时将使用它，否则使用软件来填充内存。不管是哪种情况，我们都不必关心，因为我们将编写 API 函数，用于清除辅助缓存和主缓存：

```
Fill_Primary(int color);
Fill_Secondary(int color);
```

上述函数假设缓存未被锁定，参数 color 为 8 位或 16 位（这取决于使用的颜色模式）。在大多数情况下，使用黑色来清除缓存，因此将 color 参数设置为 0。这是因为大多数调色板都使用索引 0 表示黑色，同时 USHORT 值 0 相当于 RGB(0，0，0)，也是黑色。

使用清除函数相当简单。在动画循环最开始，调用它来清屏。然而，由于在动画循环末尾将把辅助缓存复制到主缓存中，因此没有必要清除主缓存，而只需清除辅助缓存即可。

下面的范例演示了如何以 800×600×8 模式、30 帧/秒的速度，以动画方式在屏幕上随机地绘制像素：

```
UCHAR *primary_buffer; // 主缓存
int  primary_pitch; // 内存跨距，单位为字节

UCHAR *secondary_buffer; // 辅助缓存
int  seconday_pitch;  // 内存跨距，单位为字节

UCHAR pixel; // 要写入的 8 位像素
int x,y;   // 像素的坐标

// 第 0 步：创建窗口
Create_Window(800, 600, 8);

// 进入无限循环
while(1)
{
// 清除辅助缓存
Fill_Secondary(0);

// 第 1 步：锁定辅助缓存
Lock_Secondary(&secondary_buffer, &secondary_pitch);

for (int num_dots=0; num_dots < 1000; num_dots++)
  {
  // 随机生成 x、y、pixel 的值
  x = rand( )%800;
  y = rand( )%600;
  pixel = rand( )%256;

  // 将像素写入到屏幕中相应的位置
  video_buffer[x + y*secondary_pitch)] = pixel;
  } // end for num_dots

// 执行写入操作后，解除对辅助缓存的锁定
Unlock_Secondary(secondary_buffer);

// 交换缓存
Flip_Display( );

// 等待一会儿
Sleep(33);

} // end while
```

当然，这个循环是无限循环，在实际的 Windows 应用程序中，不能这样贪婪，应该在绘制每帧后将控制权交给主事件循环。

3.2.4　完整的虚拟图形系统

至此，我们有足够的功能来实现虚拟图形系统，这里总结一下我们有的功能：

- 我们假设可以调用函数，来设置图形系统以及以所需的大小和位深打开窗口（也可以是全屏）。然而，在大多数时候，我们只使用 8 位模式和 16 位模式，以提高速度。
- 系统包含主缓存和辅助离屏缓存。缓存是可线性寻址的，但是有相关联的内存跨距。另外，访问缓存的实际内存时，必须锁定缓存；完成操作后，必须解除对缓存的锁定。
- 为产生动画显示以及交换页面，我们假设有一个函数，可以利用硬件块图像传递加速来完成这项工作。然而，调用该函数前，必须确保主缓存和辅助缓存都没有被锁定。另外，还有缓存清除函数，它们尽可能地使用硬件加速。

如果能够实现这个虚拟软件系统，则编程的 3D 图形方面将是余下的惟一问题。只需与帧缓存交互，并调用几个 API 函数，就可以在屏幕上绘制图像。

3.2.5　I/O、声音和音乐

本书的主要目标是制作可玩的 3D 游戏，因此需要能够读取玩家的输入以及播放声音和音乐。

这些都是 DirectX 的功能，我编写了一些封装函数，使得它们使用起来尽可能简单。这里我要说的是，即使将本书的代码移植到不支持 DirectX 的系统中，也不需要做太多的工作。《Windows 游戏编程大师技巧》中创建的引擎只有如下基本功能：

1. 初始化并检测所有输入设备：键盘、鼠标和游戏杆。
2. 读取数据或所有输入设备的状态，并将数据存储到简单数据结构中。
3. 初始化声音和音乐系统。
4. 从磁盘上加载并播放.WAV 格式的声音文件（播放一次或循环播放）；从磁盘上加载并播放 MIDI 歌曲（播放一次或循环播放）；播放时检测声音和音乐的状态。
5. 关闭一切东西。

这些就是我们要实现的功能。然而，如果需要，可以删除任何演示程序中的 I/O 和声音代码，只移植图形接口。当然，您需要某种形式的输入，哪怕是 getch() 函数！

最终的 API 与下面类似：

```
// 初始化所有的输入设备
Init_Input_Devs( );

// 关闭输入系统
Shutdown_Input_Devs( );

// 读取键盘，数据是存储键盘状态的某种结构，可能是数组
Read_Keyboard(&data);

// 读取鼠标，数据是存储鼠标状态（可能是位置和按钮）的某种结构
Read_Mouse(&data);

// 读取游戏杆，数据是存储游戏杆状态（可能是位置和按钮）的某种结构
Read_Joystick(&data);

// 初始化声音和音乐系统
Init_Sound_Music( );

// 关闭声音和音乐系统
Shutdown_Sound_Music( );

// 加载一个.WAV 文件并返回一个 ID
int id = Load_WAV(char *filename);
```

```
// 播放 ID 指定的.WAV 文件
Play_WAV(int id);

// 加载一个.MID 文件并返回一个 ID
int id = Load_MID(char *filename);

// 播放 ID 指定的.MID 文件
Play_MID(int id);
```

如果能够实现这些函数，则有关 I/O 和声音接口的工作便全部完成了。

有关虚拟计算机接口的抽象设计就介绍到这里。别忘了，这只是一个软件设计练习而已。现在需要实现其细节，即所有函数的实际实现。

提示：这是一个非常有用的设计练习。基本上，我们设计了一个可移植的图形系统。采用这些思想，将能够设计出这样的游戏引擎，即通过实现少数几个函数的“内部细节”，就能够将它移植到其他平台。

3.3 T3DLIB 游戏控制台

现在我又面临另一个困境。我想列出通用的 Windows 游戏控制台，后面将用它构建游戏；然而，其内部全部是 API 调用。是现在介绍这些库函数 API 还是等到以后介绍呢？由于读者是中高级游戏程序员，因此可以等待。如果有什么内容不能理解，可以查看库 API 清单和描述。

3.3.1 T3DLIB 系统概述

根据前面对虚拟计算机设计的讨论，我们的目标是将 Win32/DirectX 模型抽象成一个非常简单的双缓存图形系统，我们可以访问该系统，并支持用户输入和声音。基于这个目标，我创建了 T3D 游戏控制台。该游戏控制台是创建虚拟计算机接口的第一步。处理 DirectX、输入和声音之前，首先需要从方程中消除 Windows。我将创建一个模板（游戏控制台），它使得从我们的角度看，Windows 应用程序与标准 DOS/Unix 应用程序没有什么差别。

我们将逐层地建立游戏控制台，直到实现虚拟计算机。首先我们将从方程中消除 Windows。

3.3.2 基本游戏控制台

第一个版本的游戏控制台应执行下列任务：

1. 打开窗口。
2. 调用用户定义的初始化函数：Game_Init()。
3. 进入到主 Windows 事件循环、处理消息，然后返回。
4. 调用用户定义的主工作函数 Game_Main()，它执行一次游戏逻辑循环，然后返回。
5. 回到第 3 步，直到用户关闭应用程序。
6. 调用用户定义的关闭函数 Game_Shutdown()，它执行清除工作。

图 3.7 是游戏控制台的流程图。读者可能注意到了，初始化函数、主工作函数和关闭函数分别被命名为 Game_Init()、Game_Main() 和 Game_Shutdown()。

如果能够实现一个执行步骤 1～6 中功能的外壳程序，则完全不用考虑 Windows，从而将精力集中在函数 Game_Init()、Game_Main() 和 Game_Shutdown() 上。首先，来看这 3 个函数的原型（它们的接口相同）：

```
// 游戏控制台
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);
```

实际上，我已经编写了这些函数，您可以发送一个结构到这些函数，但我从来没有使用过这种功能。我们需要编写一个 Windows 外壳程序，它完成如下任务：

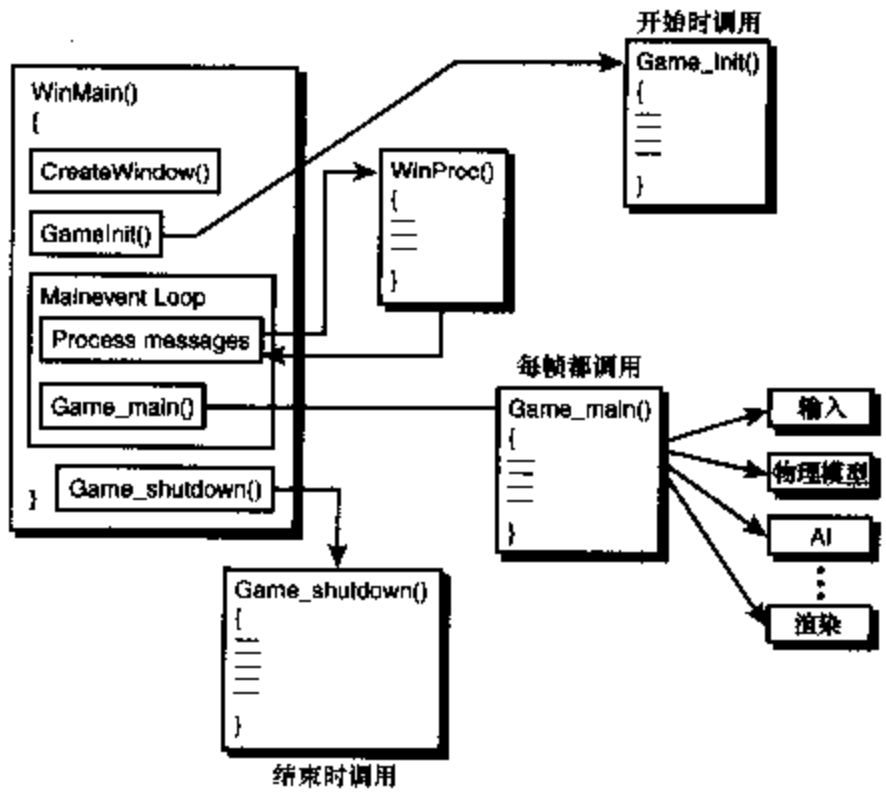


图 3.7　游戏控制台的控制流程

1．创建并注册 Windows 类。

2．创建窗口。

3．有一个事件循环和一个 WinProc()，用于处理基本消息 WM_CREATE、WM_PAINT 和 WM_DESTROY。

4．调用事件循环之前，先调用 Game_Init()，以便能够在其中进行初始化。

5．在事件循环期间，每次循环将调用 Game_Main()，让游戏能够运行。

6．窗口被关闭时，调用 Game_Shutdown()，执行清除工作。

下面是附带光盘中 T3DCONSOLEALPHA1.CPP 的代码，它是一个执行上述所有步骤的完整的 Windows 程序，其中对工作函数的调用代码为粗体：

```
// T3DCONSOLEALPHA1.CPP：第一个版本的游戏控制台

// 包含文件 ////////////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN

#include <windows.h>   // 包含重要的 Windows 头文件
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // 包含重要的 C/C++头文件
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

// 常量 ////////////////////////////////////////////////////

// 用于 Windows 接口的常量
#define WINDOW_CLASS_NAME "WIN3DCLASS"  // Windows 类的名称
#define WINDOW_TITLE    "T3D Graphics Console Alpha 1.0"
#define WINDOW_WIDTH    320  // size of window
#define WINDOW_HEIGHT    240

// 用于以异步方式读取键盘输入
```

```
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code)   ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// 函数原型 ////////////////////////////////////////////////

// 游戏控制台
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// 全局变量 ////////////////////////////////////////////////

HWND main_window_handle  = NULL; // 用于存储窗口句柄
HINSTANCE main_instance  = NULL; // 用于存储实例
char buffer[256];          // 用于打印文本


// 函数 ////////////////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
         UINT msg,
              WPARAM wparam,
              LPARAM lparam)
{
// 这是系统的主消息处理程序
PAINTSTRUCT   ps;     // 用于 WM_PAINT
HDC     hdc;      // 设备场境句柄

// 判断是什么消息
switch(msg)
  {
  case WM_CREATE:
    {
  // 初始化代码
  return(0);
  } break;

    case WM_PAINT:
    {
    // 开始绘制
    hdc = BeginPaint(hwnd,&ps);

    // 结束绘制
    EndPaint(hwnd,&ps);
    return(0);
    } break;

  case WM_DESTROY:
    {
  // 关闭应用程序
  PostQuitMessage(0);
  return(0);
  } break;

  default:break;

   } // end switch

// 处理其他消息
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////////

int WINAPI WinMain( HINSTANCE hinstance,
      HINSTANCE hprevinstance,
```

```
        LPSTR lpcmdline,
        int ncmdshow)
{
// 这是 winmain 函数

WNDCLASS winclass;  // 用于存储将创建的 Windows 类
HWND    hwnd;     // 窗口句柄
MSG     msg;    // 消息
HDC     hdc;    // 设备场境
PAINTSTRUCT ps;  // paintstruct

// 设置窗口类
winclass.style     = CS_DBLCLKS | CS_OWNDC |
                CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc   = WindowProc;
winclass.cbClsExtra    = 0;
winclass.cbWndExtra    = 0;
winclass.hInstance     = hinstance;
winclass.hIcon       = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor      = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground    = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName    = NULL;
winclass.lpszClassName    = WINDOW_CLASS_NAME;

// 注册窗口类
if (!RegisterClass(&winclass))
  return(0);

// 创建窗口，根据 WINDOWED_APP 是否为 true 选择合适的窗口标记
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // 窗口类
        WINDOW_TITLE,    // 标题
             WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION,
         0,0,   // 位置
        WINDOW_WIDTH, // 宽度
               WINDOW_HEIGHT, // 高度
        NULL,     // 父窗口句柄
        NULL,     // 菜单句柄
        hinstance,// 实例
        NULL)))  // 创建参数
return(0);

// 将窗口句柄和实例存储到全局变量中
main_window_handle = hwnd;
main_instance      = hinstance;

// 显示窗口
ShowWindow(main_window_handle, SW_SHOW);

// 执行游戏控制台特有的初始化
Game_Init( );

// 进入主事件循环
while(1)
  {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
    // 检查是否是退出消息
     if (msg.message == WM_QUIT)
       break;

    // 转换加速键
    TranslateMessage(&msg);

    // 将消息发送给 window proc
    DispatchMessage(&msg);
     } // end if
```

```
  // 主游戏处理函数
  Game_Main( );
  } // end while

// 关闭游戏并释放所有的资源
Game_Shutdown( );

// 返回到 Windows 操作系统
return(msg.wParam);

} // end WinMain

// T3DII 游戏编程控制台函数 ////////////////

int Game_Init(void *parms)
{
// 所有的游戏初始化工作都在这个函数中执行

// 成功返回
return(1);

} // end Game_Init

//////////////////////////////////////////////////////////

int Game_Shutdown(void *parms)
{
// 在这个函数中关闭游戏并释放为游戏分配的所有资源

// 成功返回
return(1);
} // end Game_Shutdown

//////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
// 这是游戏的核心，将不断地被实时调用
// 它类似于 C 语言中的 main( )，所有游戏调用都是在这里进行的

// 游戏逻辑

// 检查用户是否要退出
if (KEY_DOWN(VK_ESCAPE))
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);
  } // end if

// 成功返回
return(1);

} // end Game_Main
```

在附带光盘中，上述源代码文件名为 T3DCONSOLEALPHA1.CPP，可执行文件名为 T3DCONSOLEALPHA1.EXE。运行这个可执行文件时，将看到如图 3.8 所示的图像：一个小窗口，其中没有任何东西。然而，在幕后实际上执行了很多操作：创建一个窗口、调用 Game_Init() 函数，在每次事件循环中调用 Game_Main() 函数。最后，用户关闭该窗口时，将调用 Game_Shutdown() 函数。

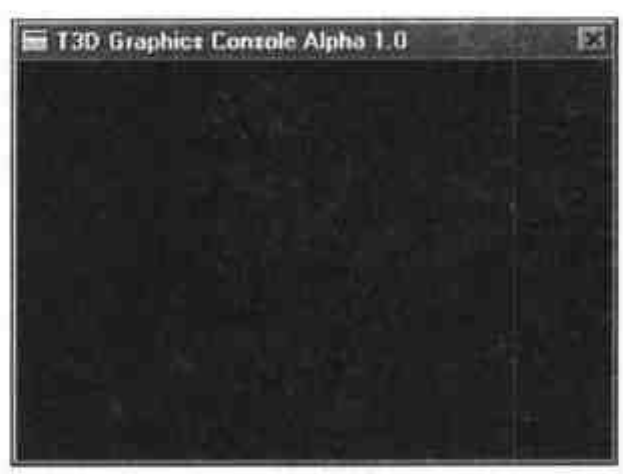


图 3.8　第 1 版游戏控制台运行时的屏幕截图

您只需将全部游戏功能加入到这 3 个函数中，仅此而已——我们已经将一个 Windows 应用程序抽象成 3 个函数调用，其他工作都已经完成。

这个第一版的游戏控制台提供了所需的全部 Windows 功能。现在需要 DirectX 接口来模拟虚拟计算机，并提供基本的双缓存图形系统以及声音和输入功能。现在，可以直接列出最终版本的游戏控制台代码，它具备上述功能，但更好的方法是先来看一下用来创建最终游戏控制台的 API 函数。您无需完全理解这些 API 函数，只需了解这些 API 及其相关解释和范例。

查看用于创建最终游戏控制台的工具后，我们将以 T3DCONSOLEALPHA1.CPP 文件为基础，在其中添加各种功能，使其成为我们需要的虚拟计算机接口。本书所有的演示程序和游戏都以这个模板为起点。另外，本章最后将通过一些范例，演示如何使用 T3DLIB 的图形、声音和输入功能。下面介绍《Windows 游戏编程大师技巧》中创建的 T3DLIB 游戏引擎的 3 个子系统。

3.4　T3DLIB1 库

来看一下 T3DLIB 游戏库的图形模块（T3DLIB1.CPP）中的#define、宏、数据结构和函数。

注意：如果您阅读过《Windows 游戏编程大师技巧》，将发现该引擎除 16 位支持和窗口支持外，几乎与前一个引擎完全相同。其代码仍然是兼容的，可以用新版本的 T3DLIB1.CPP 来编译《Windows 游戏编程大师技巧》中的演示程序，它们仍可以正常工作！当然，所有内容都将用 DirectX 8.0 或 9.0+编译。

该模块由两个文件组成：T3DLIB1.CPP|H。因此，只需将这两个文件链接到程序中，便可以使用这些 API 函数。

3.4.1　DirectX 图形引擎体系结构

如图 3.9 所示，T3DLIB1 是一个相当简单的 2D 引擎。实质上，它是一个 2D、8/16 位颜色、双缓存的 DirectX 引擎，支持任何分辨率和裁剪。该引擎还支持窗口模式和全屏模式，并负责所有设置工作。因此，不管是在窗口模式下还是在全屏模式下，都可以将像素写入辅助缓存，程序逻辑负责将辅助缓存复制到主缓存的处理细节。

要使用这个库来创建应用程序，需要在程序中包含 T3DLIB1.CPP|H、DDRAW.LIB（DirectDraw 库）和 WINMM.LIB（Win32 多媒体库）。

注意：仅当使用的是 Visual C++时，才需要包含 WINMM.LIB。

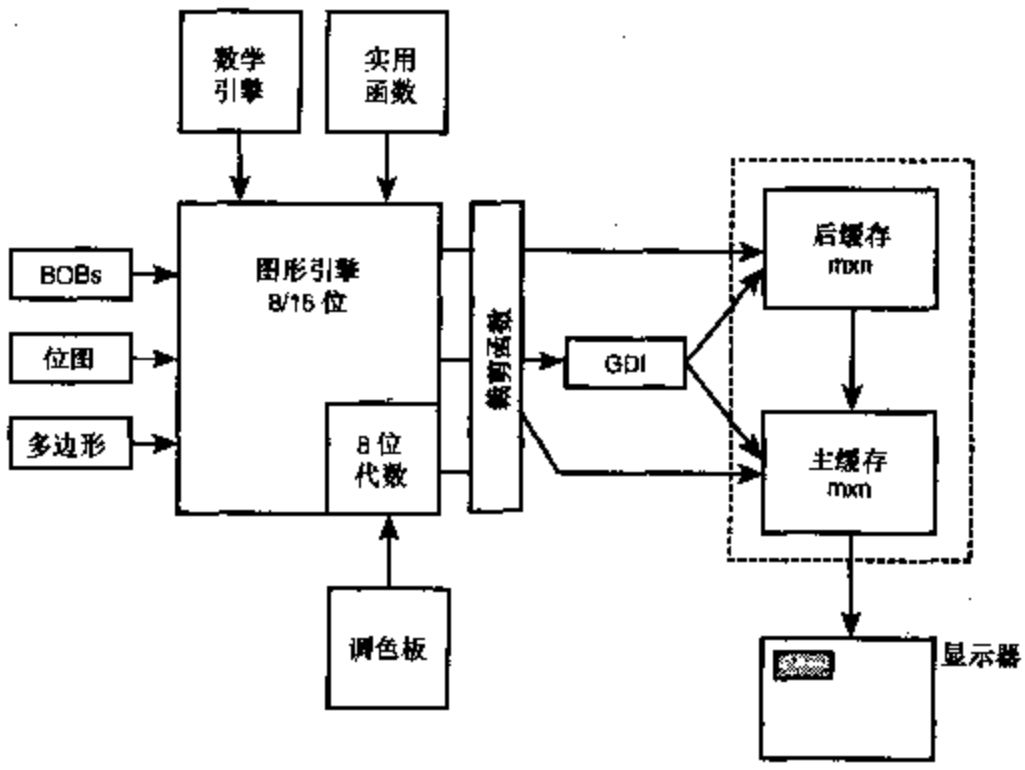


图 3.9　图形引擎的体系结构

3.4.2　基本常量

该引擎有一个头文件 T3DLIB1.H，其中定义了很多引擎要使用的常量。下面列出了这些常量，供读者参考：

```
// 避免重复包含
#ifndef T3DLIB1
#define T3DLIB1

// 常量 ////////////////////////////////////////////////////

// 默认屏幕设置，调用函数 DDraw_Init( )，这些值将被覆盖
// 这里定义它们，旨在用于给全局常量赋值
#define SCREEN_WIDTH    640  // 屏幕的大小
#define SCREEN_HEIGHT    480
#define SCREEN_BPP      8   // 每个像素占多少位
#define MAX_COLORS_PALETTE  256


#define DEFAULT_PALETTE_FILE "PALDATA2.PAL"
// 用于选择窗口/全屏模式
#define SCREEN_FULLSCREEN   0
#define SCREEN_WINDOWED    1

// 与位图相关的常量
#define BITMAP_ID       0x4D42 // 位图的全局 ID
#define BITMAP_STATE_DEAD  0
#define BITMAP_STATE_ALIVE  1
#define BITMAP_STATE_DYING  2
#define BITMAP_ATTR_LOADED  128

#define BITMAP_EXTRACT_MODE_CELL  0
#define BITMAP_EXTRACT_MODE_ABS   1

// directdraw 像素格式常量
// 用于帮助位图加载函数将数据设置为正确的格式
#define DD_PIXEL_FORMAT8     8
#define DD_PIXEL_FORMAT555    15
#define DD_PIXEL_FORMAT565    16
#define DD_PIXEL_FORMAT888     24
#define DD_PIXEL_FORMATALPHA888 32

// 针对 BOB 的常量
#define BOB_STATE_DEAD      0  // 已死
#define BOB_STATE_ALIVE     1  // 活着
#define BOB_STATE_DYING     2  // 要死了
#define BOB_STATE_ANIM_DONE  1    // 动画结束状态
#define MAX_BOB_FRAMES      64    // 最大 BOB 帧数
#define MAX_BOB_ANIMATIONS   16    // 最大动作数

#define BOB_ATTR_SINGLE_FRAME   1    // BOB 为单帧
#define BOB_ATTR_MULTI_FRAME   2    // BOB 包含多帧
#define BOB_ATTR_MULTI_ANIM    4    // BOB 包含多个动作
#define BOB_ATTR_ANIM_ONE_SHOT  8    // BOB 将执行动作一次
#define BOB_ATTR_VISIBLE     16  // BOB 可见
#define BOB_ATTR_BOUNCE      32  // BOB 反弹
#define BOB_ATTR_WRAPAROUND    64  // BOB 紧贴彼边
#define BOB_ATTR_LOADED      128 // BOB 已加载
#define BOB_ATTR_CLONE      256 // BOB 是副本

// 屏幕渐变命令（只适用于 256 颜色模式）
#define SCREEN_DARKNESS 0    // 渐变为黑色
#define SCREEN_WHITENESS 1     // 渐变为白色
#define SCREEN_SWIPE_X  2     // 水平扫描
#define SCREEN_SWIPE_Y  3     // 垂直扫描
#define SCREEN_DISOLVE  4     // 像素淡出
#define SCREEN_SCRUNCH  5     // 方块压缩
#define SCREEN_BLUENESS 6     // 渐变为蓝色
#define SCREEN_REDNESS  7     // 渐变为红色
#define SCREEN_GREENNESS 8      // 渐变为绿色

// 用于 Blink_Colors 的常量
```

```
#define BLINKER_ADD      0   // 将光源加入到数据库中
#define BLINKER_DELETE    1   // 从数据库中删除光源
#define BLINKER_UPDATE    2   // 更新光源
#define BLINKER_RUN      3   // 正常运行

// 圆周率常量
#define PI      ((float)3.141592654f)
#define PI2     ((float)6.283185307f)
#define PI_DIV_2   ((float)1.570796327f)
#define PI_DIV_4   ((float)0.785398163f)
#define PI_INV    ((float)0.318309886f)

// 定点数学常量
#define FIXP16_SHIFT    16
#define FIXP16_MAG     65536
#define FIXP16_DP_MASK   0x0000ffff
#define FIXP16_WP_MASK   0xffff0000
#define FIXP16_ROUND_UP  0x00008000
```

3.4.3　工作宏

接下来是我们已经编写的全部宏：

```
// 以异步方式读取键盘输入
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
#define KEY_UP(vk_code)   ((GetAsyncKeyState(vk_code) & 0x8000) ? 0 : 1)

// 生成 5.5.5 格式的 16 位颜色值
#define _RGB16BIT555(r,g,b) ((b & 31) + ((g & 31) << 5) + ((r & 31) << 10))

//生成 5.6.5 格式的 16 位颜色值
#define _RGB16BIT565(r,g,b) ((b & 31) + ((g & 63) << 5) + ((r & 31) << 11))

// 生成 8.8.8 格式的 24 位颜色值
#define _RGB24BIT(a,r,g,b) ((b) + ((g) << 8) + ((r) << 16) )

// 生成 A.8.8.8 格式的 32 位颜色值
#define _RGB32BIT(a,r,g,b) ((b) + ((g) << 8) + ((r) << 16) + ((a) << 24))

// 位运算宏
#define SET_BIT(word,bit_flag)   ((word)=((word) | (bit_flag)))
#define RESET_BIT(word,bit_flag) ((word)=((word) & (~bit_flag)))

// 初始化一个 direct draw 结构，基本上清空其内容并设置字段 dwSize
#define DDRAW_INIT_STRUCT(ddstruct) {
memset(&ddstruct,0,sizeof(ddstruct)); ddstruct.dwSize=sizeof(ddstruct); }

// 用于计算两个表达式中较大和较小的一个
#define MIN(a, b)  (((a) < (b)) ? (a) : (b))
#define MAX(a, b)  (((a) > (b)) ? (a) : (b))

// 用于交换两个变量的值
#define SWAP(a,b,t) {t=a; a=b; b=t;}

// 一些数学宏
#define DEG_TO_RAD(ang) ((ang)*PI/180.0)
#define RAD_TO_DEG(rads) ((rads)*180.0/PI)

#define RAND_RANGE(x,y) ( (x) + (rand( )%((y)-(x)+1)))
```

3.4.4　数据类型和结构

接下来是引擎需要使用的数据类型和数据结构。其中很多对于本书意义不大，因为它们用于《Windows 游戏编程大师技巧》的 2D 方面，但查看全部数据类型和数据结构也没有什么坏处。

```
// 基本无符号类型
```

```
typedef unsigned short  USHORT;
typedef unsigned short  WORD;
typedef unsigned char  UCHAR;
typedef unsigned char  BYTE;
typedef unsigned int   QUAD;
typedef unsigned int   UINT;

// .BMP 文件位图的容器结构
typedef struct BITMAP_FILE_TAG
{
BITMAPFILEHEADER bitmapfileheader;  // 包含位图文件头
BITMAPINFOHEADER bitmapinfoheader;  // 包括调色板在内的所有信息
PALETTEENTRY   palette[256];    // 存储调色板
UCHAR       *buffer;      // 指向数据的指针

    } BITMAP_FILE, *BITMAP_FILE_PTR;

// BOB
typedef struct BOB_TYP
    {
    int state;     // 状态
    int anim_state;    // 动作状态变量
    int attr;      // 属性
    float x,y;     // 位置
    float xv,yv;    // 速度
    int width, height;  // 宽度和高度
    int width_fill;     // 用于强制表面的宽度为 8 的倍数
    int bpp;       // 每个像素占用的位数
    int counter_1;     // 计数器
    int counter_2;
    int max_count_1;    // 阈值
    int max_count_2;
    int varsI[16];     // 深度为 16 的整数栈
    float varsF[16];    // 深度为 16 的浮点数栈
    int curr_frame;     // 当前动画帧
    int num_frames;     // 总动画帧数
    int curr_animation; // 当前动作索引
    int anim_counter;   // 用于给动作切换定时
    int anim_index;     // 动作元素索引
    int anim_count_max; // 几帧后再切换动作
    int *animations[MAX_BOB_ANIMATIONS]; // 动作

    LPDIRECTDRAWSURFACE7 images[MAX_BOB_FRAMES]; // 位图图像 DD 表面
    } BOB, *BOB_PTR;

// 简单位图图像
typedef struct BITMAP_IMAGE_TYP
    {
    int state;       // 状态
    int attr;        // 属性
    int x,y;        // 位置
    int width, height;  // 尺寸
    int num_bytes;    // 总字节数
    int bpp;      // 每个像素占用多少位
    UCHAR *buffer;    // 位图中的像素数

    } BITMAP_IMAGE, *BITMAP_IMAGE_PTR;

// 闪光灯结构
typedef struct BLINKER_TYP
    {
    // 由用户设置
    int color_index;      // 不考虑的颜色索引
    PALETTEENTRY on_color;   // 开时颜色的 RGB 值
    PALETTEENTRY off_color;  // 关时颜色的 RGB 值
    int on_time;        // 持续开多少帧
    int off_time;      // 持续关多少帧
```

```
    // 内部成员
    int counter;         // 状态切换计数器
    int state;          // 光源状态，1 表示开，-1 表示关，0 表示无效
    } BLINKER, *BLINKER_PTR;

// 2D 顶点
typedef struct VERTEX2DI_TYP
    {
    int x,y; // 顶点坐标
    } VERTEX2DI, *VERTEX2DI_PTR;

// 2D 顶点
typedef struct VERTEX2DF_TYP
    {
    float x,y; // 顶点坐标
    } VERTEX2DF, *VERTEX2DF_PTR;

// 2D 多边形
typedef struct POLYGON2D_TYP
    {
    int state;     // 状态
    int num_verts;  // 顶点数
    int x0,y0;     // 多边形中心的位置
    int xv,yv;     // 初始速度
    DWORD color;    // 索引或 PALETTENTRY
    VERTEX2DF *vlist; // 指向顶点列表的指针

    } POLYGON2D, *POLYGON2D_PTR;


// 矩阵定义
typedef struct MATRIX3X3_TYP
    {
    union
    {
    float M[3][3]; // 以数组方式存储

    // 按先行后列的顺序以单独变量方式存储
    struct
       {
       float M00, M01, M02;
       float M10, M11, M12;
       float M20, M21, M22;
       }; // end explicit names

    }; // end union
    } MATRIX3X3, *MATRIX3X3_PTR;

typedef struct MATRIX1X3_TYP
    {
    union
    {
    float M[3]; // 以数组方式存储

    // 按先行后列的顺序以单独变量方式存储
    struct
       {
       float M00, M01, M02;

       }; // end explicit names
    }; // end union
    } MATRIX1X3, *MATRIX1X3_PTR;

typedef struct MATRIX3X2_TYP
    {
```

```
    union
    {
    float M[3][2]; // 以数组方式存储

    // 按先行后列的顺序以单独变量方式存储
    struct
       {
       float M00, M01;
       float M10, M11;
       float M20, M21;
       }; // end explicit names

    }; // end union
    } MATRIX3X2, *MATRIX3X2_PTR;

typedef struct MATRIX1X2_TYP
    {
    union
    {
    float M[2]; // 以数组方式存储

    // 按先行后列的顺序以单独变量方式存储
    struct
       {
       float M00, M01;

       }; // end explicit names
    }; // end union
    } MATRIX1X2, *MATRIX1X2_PTR;
```

最后是一些数学结构，它们是《Windows 游戏编程大师技巧》中的 2D 多边形变换支持。创建新的 3D 数学库时我们将重新编写这部分内容。

3.4.5 函数原型

接下来看一下完整的函数原型列表，在后面的几节中将不会介绍函数 API 中的所有函数，因为很多函数都只是另一个函数的不同版本。

```
// DirectDraw 函数
int DDraw_Init(int width, int height, int bpp, int windowed=0);
int DDraw_Shutdown(void);
LPDIRECTDRAWCLIPPER DDraw_Attach_Clipper(LPDIRECTDRAWSURFACE7 lpdds,
                    int num_rects, LPRECT clip_list);

LPDIRECTDRAWSURFACE7 DDraw_Create_Surface(int width, int height,
                  int mem_flags=0, USHORT color_key_value=0);

int DDraw_Flip(void);
int DDraw_Wait_For_Vsync(void);

int DDraw_Fill_Surface(LPDIRECTDRAWSURFACE7 lpdds, USHORT color,
           RECT *client=NULL);

UCHAR *DDraw_Lock_Surface(LPDIRECTDRAWSURFACE7 lpdds,int *lpitch);
int DDraw_Unlock_Surface(LPDIRECTDRAWSURFACE7 lpdds);
UCHAR *DDraw_Lock_Primary_Surface(void);
int DDraw_Unlock_Primary_Surface(void);
UCHAR *DDraw_Lock_Back_Surface(void);
int DDraw_Unlock_Back_Surface(void);

// BOB 函数
int Create_BOB(BOB_PTR bob,int x, int y,int width, int height,
        int num_frames,int attr,
        int mem_flags=0, USHORT color_key_value=0, int bpp=8);

int Clone_BOB(BOB_PTR source, BOB_PTR dest);
```

```
int Destroy_BOB(BOB_PTR bob);
int Draw_BOB(BOB_PTR bob, LPDIRECTDRAWSURFACE7 dest);

int Draw_Scaled_BOB(BOB_PTR bob, int swidth, int sheight,
        LPDIRECTDRAWSURFACE7 dest);

int Draw_BOB16(BOB_PTR bob, LPDIRECTDRAWSURFACE7 dest);

int Draw_Scaled_BOB16(BOB_PTR bob, int swidth, int sheight,
         LPDIRECTDRAWSURFACE7 dest);

int Load_Frame_BOB(BOB_PTR bob, BITMAP_FILE_PTR bitmap,
        int frame, int cx,int cy,int mode);

int Load_Frame_BOB16(BOB_PTR bob, BITMAP_FILE_PTR bitmap, int frame,
         int cx,int cy,int mode);

int Animate_BOB(BOB_PTR bob);
int Move_BOB(BOB_PTR bob);

int Load_Animation_BOB(BOB_PTR bob, int anim_index,
          int num_frames, int *sequence);

int Set_Pos_BOB(BOB_PTR bob, int x, int y);
int Set_Vel_BOB(BOB_PTR bob,int xv, int yv);
int Set_Anim_Speed_BOB(BOB_PTR bob,int speed);
int Set_Animation_BOB(BOB_PTR bob, int anim_index);
int Hide_BOB(BOB_PTR bob);
int Show_BOB(BOB_PTR bob);
int Collision_BOBS(BOB_PTR bob1, BOB_PTR bob2);

// 实用函数
DWORD Get_Clock(void);
DWORD Start_Clock(void);
DWORD Wait_Clock(DWORD count);

int Collision_Test(int x1, int y1, int w1, int h1,
         int x2, int y2, int w2, int h2);

int Color_Scan(int x1, int y1, int x2, int y2,
        UCHAR scan_start, UCHAR scan_end,
        UCHAR *scan_buffer, int scan_lpitch);

int Color_Scan16(int x1, int y1, int x2, int y2,
         USHORT scan_start, USHORT scan_end,
         UCHAR *scan_buffer, int scan_lpitch);

// 图形函数
int Draw_Clip_Line(int x0,int y0, int x1, int y1, int color,
         UCHAR *dest_buffer, int lpitch);

int Draw_Clip_Line16(int x0,int y0, int x1, int y1, int color,
          UCHAR *dest_buffer, int lpitch);

int Clip_Line(int &x1,int &y1,int &x2, int &y2);

int Draw_Line(int xo, int yo, int x1,int y1, int color,
      UCHAR *vb_start,int lpitch);

int Draw_Line16(int xo, int yo, int x1,int y1, int color,
        UCHAR *vb_start,int lpitch);

int Draw_Pixel(int x, int y,int color,
        UCHAR *video_buffer, int lpitch);

int Draw_Rectangle(int x1, int y1, int x2, int y2,
        int color,LPDIRECTDRAWSURFACE7 lpdds);
```

```
void HLine(int x1,int x2,int y,int color, UCHAR *vbuffer, int lpitch);
void VLine(int y1,int y2,int x,int color, UCHAR *vbuffer, int lpitch);
void HLine16(int x1,int x2,int y,int color, UCHAR *vbuffer, int lpitch);
void VLine16(int y1,int y2,int x,int color, UCHAR *vbuffer, int lpitch);
void Screen_Transitions(int effect, UCHAR *vbuffer, int lpitch);
int Draw_Pixel(int x, int y,int color,UCHAR *video_buffer, int lpitch);
int Draw_Pixel16(int x, int y,int color,UCHAR *video_buffer, int lpitch);

// 调色板函数
int Set_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Get_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Load_Palette_From_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette_To_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette(LPPALETTEENTRY sav_palette);
int Set_Palette(LPPALETTEENTRY set_palette);
int Rotate_Colors(int start_index, int end_index);
int Blink_Colors(int command, BLINKER_PTR new_light, int id);

// 简单位图图像函数
int Create_Bitmap(BITMAP_IMAGE_PTR image, int x, int y,
        int width, int height, int bpp=8);

int Destroy_Bitmap(BITMAP_IMAGE_PTR image);

int Draw_Bitmap(BITMAP_IMAGE_PTR source_bitmap,UCHAR *dest_buffer,
       int lpitch, int transparent);

int Draw_Bitmap16(BITMAP_IMAGE_PTR source_bitmap,UCHAR *dest_buffer,
        int lpitch, int transparent);

int Load_Image_Bitmap(BITMAP_IMAGE_PTR image,BITMAP_FILE_PTR bitmap,
          int cx,int cy,int mode);

int Load_Image_Bitmap16(BITMAP_IMAGE_PTR image,BITMAP_FILE_PTR bitmap,
          int cx,int cy,int mode);

int Scroll_Bitmap(BITMAP_IMAGE_PTR image, int dx, int dy=0);

int Copy_Bitmap(BITMAP_IMAGE_PTR dest_bitmap, int dest_x, int dest_y,
       BITMAP_IMAGE_PTR source_bitmap, int source_x, int source_y,
       int width, int height);

int Flip_Bitmap(UCHAR *image, int bytes_per_line, int height);

// 位图文件函数
int Load_Bitmap_File(BITMAP_FILE_PTR bitmap, char *filename);
int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap);

// GDI 函数
int Draw_Text_GDI(char *text, int x,int y,
        COLORREF color, LPDIRECTDRAWSURFACE7 lpdds);

int Draw_Text_GDI(char *text, int x,int y,
        int color, LPDIRECTDRAWSURFACE7 lpdds);

// 错误函数
int Open_Error_File(char *filename, FILE *fp_override=NULL);
int Close_Error_File(void);
int Write_Error(char *string, ...);

// 2d、8/16 位三角形渲染函数
void Draw_Top_Tri(int x1,int y1,int x2,int y2, int x3,int y3,
        int color,UCHAR *dest_buffer, int mempitch);

void Draw_Bottom_Tri(int x1,int y1, int x2,int y2, int x3,int y3,
          int color,UCHAR *dest_buffer, int mempitch);
```

```
void Draw_Top_Tri16(int x1,int y1,int x2,int y2, int x3,int y3,
         int color,UCHAR *dest_buffer, int mempitch);

void Draw_Bottom_Tri16(int x1,int y1, int x2,int y2, int x3,int y3,
           int color,UCHAR *dest_buffer, int mempitch);

void Draw_Top_TriFP(int x1,int y1,int x2,int y2, int x3,int y3,
         int color,UCHAR *dest_buffer, int mempitch);

void Draw_Bottom_TriFP(int x1,int y1, int x2,int y2, int x3,int y3,
           int color,UCHAR *dest_buffer, int mempitch);

void Draw_Triangle_2D(int x1,int y1,int x2,int y2,int x3,int y3,
          int color,UCHAR *dest_buffer, int mempitch);

void Draw_Triangle_2D16(int x1,int y1,int x2,int y2,int x3,int y3,
           int color,UCHAR *dest_buffer, int mempitch);

void Draw_TriangleFP_2D(int x1,int y1,int x2,int y2,int x3,int y3,
           int color,UCHAR *dest_buffer, int mempitch);

inline void Draw_QuadFP_2D(int x0,int y0,int x1,int y1,
             int x2,int y2,int x3, int y3,
             int color,UCHAR *dest_buffer, int mempitch);


// 通用 2D、8/16 位多边形渲染和变换函数
void Draw_Filled_Polygon2D(POLYGON2D_PTR poly, UCHAR *vbuffer, int mempitch);
void Draw_Filled_Polygon2D16(POLYGON2D_PTR poly, UCHAR *vbuffer, int mempitch);
int Translate_Polygon2D(POLYGON2D_PTR poly, int dx, int dy);
int Rotate_Polygon2D(POLYGON2D_PTR poly, int theta);
int Scale_Polygon2D(POLYGON2D_PTR poly, float sx, float sy);
void Build_Sin_Cos_Tables(void);
int Translate_Polygon2D_Mat(POLYGON2D_PTR poly, int dx, int dy);
int Rotate_Polygon2D_Mat(POLYGON2D_PTR poly, int theta);
int Scale_Polygon2D_Mat(POLYGON2D_PTR poly, float sx, float sy);
int Draw_Polygon2D(POLYGON2D_PTR poly, UCHAR *vbuffer, int lpitch);
int Draw_Polygon2D16(POLYGON2D_PTR poly, UCHAR *vbuffer, int lpitch);

// 数学函数
int Fast_Distance_2D(int x, int y);
float Fast_Distance_3D(float x, float y, float z);

// collision detection functions
int Find_Bounding_Box_Poly2D(POLYGON2D_PTR poly,
              float &min_x, float &max_x,
              float &min_y, float &max_y);

int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
         MATRIX3X2_PTR mb,
         MATRIX1X2_PTR mprod);

int Mat_Mul_1X3_3X3(MATRIX1X3_PTR ma,
         MATRIX3X3_PTR mb,
         MATRIX1X3_PTR mprod);

int Mat_Mul_3X3(MATRIX3X3_PTR ma,
        MATRIX3X3_PTR mb,
        MATRIX3X3_PTR mprod);

inline int Mat_Init_3X2(MATRIX3X2_PTR ma,
            float m00, float m01,
            float m10, float m11,
            float m20, float m21);
```

```
// 内存操作函数
inline void Mem_Set_WORD(void *dest, USHORT data, int count);
inline void Mem_Set_QUAD(void *dest, UINT  data, int count);
```

从上面可知，所有光栅化函数都接受一个 UCHAR*参数，它是指向帧缓存的指针。另外，所有内存跨距参数总以字节为单位。这是我们所做的规定。另外，90%的库函数都支持 16 位图形，但有些只支持 8 位颜色。最后，有很多 2D 多边形函数，这些函数都来自《Windows 游戏编程大师技巧》，虽然在本书中也可以使用它们，但我们将重新编写，因为它们是 3D 图形光栅化处理中的一部分，我将考虑使用新的名称。

3.4.6 全局变量

我喜欢全局变量，因为它们的速度非常快。另外，它们也适合于系统级变量（任何游戏引擎都有很多系统级变量），所以这里将介绍该引擎的全局变量，对这些变量进行了注释以帮助理解。

```
FILE *fp_error;            // 通用错误文件
char error_filename[80];       // 错误文件名

// 使用的很多接口是 7.0 版的
LPDIRECTDRAW7     lpdd;      // dd 对象
LPDIRECTDRAWSURFACE7 lpddsprimary;  // dd 主缓存
LPDIRECTDRAWSURFACE7 lpddsback;   // dd 后缓存
LPDIRECTDRAWPALETTE lpddpal;     // 指向 dd 调色板的指针
LPDIRECTDRAWCLIPPER lpddclipper;   // 用于后缓存的 dd 裁剪程序
LPDIRECTDRAWCLIPPER lpddclipperwin; // 用于窗口的 dd 裁剪程序
PALETTEENTRY     palette[256];  // 调色板
PALETTEENTRY     save_palette[256]; // 用于保存调色板
DDSURFACEDESC2    ddsd;   // dd 面描述结构
DDBLTFX        ddbltfx; // 用于填充
DDSCAPS2        ddscaps; // dd 面功能结构
HRESULT       ddrval;  // dd 调用返回的结果
UCHAR        *primary_buffer; // 主视频缓存
UCHAR        *back_buffer;  // 辅助缓存
int          primary_lpitch; // 内存行跨距
int          back_lpitch;   // 内存行跨距
BITMAP_FILE    bitmap8bit;   // 8 位的位图文件
BITMAP_FILE    bitmap16bit;   // 16 位的位图文件
BITMAP_FILE    bitmap24bit;   // 24 位的位图文件

DWORD start_clock_count; // 用于计时
int  windowed_mode;      // 记录 dd 是否处于窗口模式

// 这些变量定义了软件裁剪时使用的裁剪矩形
int min_clip_x, // 裁剪矩形
  max_clip_x,
  min_clip_y,
  max_clip_y;

// 调用 DD_Init( )时将覆盖这些变量的值
int screen_width,  // 屏幕宽度
  screen_height,  // 屏幕高度
  screen_bpp,   // 每个像素占用多少位
  screen_windowed; // 是窗口模式应用程序吗?

int dd_pixel_format; // 默认像素格式

int window_client_x0; // 用于存储 dd 窗口模式，客户区域的左上角的位置
int window_client_y0;

// 用于存储查找表
float cos_look[360];
float sin_look[360];

// 指向 RGB16 颜色生成函数的指针，这两个函数分别生成 5.5.5 和 5.6.5 格式的颜色值
USHORT (*RGB16Bit)(int r, int g, int b);
```

请注意 DirectX 接口的全局名称以及定义裁剪矩形和屏幕大小的全局变量。

3.4.7　DirectDraw 接口

介绍所有的数据支持后，下面来看一看所有的 DirectDraw 支持函数。《Windows 游戏编程大师技巧》中的 DirectDraw 系统有下列功能：

- 双缓存（主缓存和辅助缓存）；
- 采用调色板的 8 位颜色模式；
- 采用自动像素格式检测的 16 位颜色模式；
- 8 位和 16 位颜色模式下的窗口支持；
- 2D 多边形和光栅位图裁剪；
- 访问主缓存和辅助缓存；
- 用于窗口显示的页面交换和缓存复制。

注意：请注意最后一项功能。DirectX 不允许窗口应用程序使用双缓存（在 DirectX 中称为复合面[complex surface]）。因此，请求窗口显示时，程序不会创建标准的全屏复合面（使用主缓存和后备缓存），而是创建一个主缓存（它为整个屏幕）和一个离屏缓存（其大小与窗口客户区域相同）。然后请求执行换页时，程序将把离屏后缓存复制到窗口的客户区域。当然，所有这些对于您来说都是透明的。

下面介绍每个函数并列举范例。

函数原型：

```
int DDraw_Init(int width, // 显示区域的宽度
        int  height, // 显示区域的高度
        int  bpp,  // 每个像素占用多少位
        int  windowed=0); // 0 表示全屏幕，1 表示窗口模式
```

用途：

DDraw_Init()用于设置和初始化 DirectDraw。您可以使用任何分辨率和色深。如果要使用窗口显示，将最后一个参数设置为 1；默认为全屏显示。如果成功执行，返回 TRUE。

范例：

```
// 将系统设置为全屏模式、800x600、256 色
DDraw_Init(800,600,8);
// 将系统设置为窗口模式、400x400、16 位颜色模式
// 在这种情况下，桌面必须为 16 位颜色模式
DDraw_Init(400,400,16,1);
```

函数原型：

```
int DDraw_Shutdown(void);
```

用途：

DDraw_Shutdown()关闭 DirectDraw 并释放所有接口。

范例：

```
// 在程序中，关闭代码可能如下
DDraw_Shutdown( );
```

函数原型：

```
LPDIRECTDRAWCLIPPER
 DDraw_Attach_Clipper(
  LPDIRECTDRAWSURFACE7 lpdds, // 要关联到哪个表面（即缓存——译者注）
  int num_rects,   // 矩形数
  LPRECT clip_list); // 指向矩形的指针
```

用途：

DDraw_Attach_Clipper()将一个裁剪区域与指定表面（通常为后缓存）关联起来。另外，还必须指定裁剪矩形列表及其包含的矩形数量。如果成功执行，则返回 TRUE。

范例：

```
// 创建一个大小与屏幕相同的裁剪区域
RECT clip_zone = {0,0,SCREEN_WIDTH-1, SCREEN_HEIGHT-1};
DDraw_Attach_Clipper(lpddsback, 1, &clip_zone);
```

函数原型：

```
LPDIRECTDRAWSURFACE7
  DDraw_Create_Surface(int width, // 表面宽度
       int height,     // 表面高度
       int mem_flags=0,   // 控制标记
       USHORT color_key_value=0); // 颜色键值
```

用途：

DDraw_Create_Surface()用于在系统内存、VRAM 或 AGP 内存中创建一个常规离屏 DirectDraw 表面。默认为 DDSCAPS_OFFSCREENPLAIN。可以使用逻辑 OR 将任何控制标与默认值组合起来。它们是标准的 DirectDraw DDSCAP*标记，例如，DDSCAPS_SYSTEMMEMORY 和 DDSCAPS_VIDEOMEMORY 分别代表系统内存和 VRAM。该函数使用其内部逻辑，根据前面设置的图形模式，创建 8 位或 16 位颜色表面。最后一个参数用于指定颜色键值，默认为（int）0。在 8 位颜色模式下，它相当于索引 0；在 16 位颜色模式下，它相当于 RGB(0，0，0)。要将其他值用作透明颜色键值，可相应地设置该参数，以覆盖默认值。如果该函数成功执行，将返回指向新表面的指针；否则返回 NULL。

范例：

```
// 在 VRAM 中创建一个 64×64 的表面
LPDIRECTDRAWSURFACE7 image = DDraw_Create_Surface(64,64, DDSCAPS_VIDEOMEMORY);

// 在离屏内存中创建一个 28×128 的表面，并将颜色键值设置为 16
// 这里假设颜色模式位 8 位
LPDIRECTDRAWSURFACE7 image =
       DDraw_Create_Surface(128,128, DDSCAPS_OFFSCREENPLAIN ,16);
```

函数原型：

```
int DDraw_Flip(void);
```

用途：

在全屏模式下，DDraw_Flip()将辅助表面复制到主表面；在窗口模式下，它将虚拟辅助缓存复制到窗口的客户区域。该函数将一直等待，直到可以执行复制，因此它可能不会立即返回。如果成功执行，则返回 TRUE。

范例：

```
DDraw_Flip( );
```

函数原型：

```
int DDraw_Wait_For_Vsync(void);
```

用途：

DDraw_Wait_For_Vsync（void）将等待，直到下一个垂直空白间隔开始（光栅到达屏幕底部）。如果成功执行，则返回 TRUE，否则返回 FALSE。

范例：

```
// 等待 1/70 秒
DDraw_Wait_For_Vsync( );
```

函数原型：

```
int DDraw_Fill_Surface(LPDIRECTDRAWSURFACE7 lpdds, // 要填充的表面
            int color, // 用于填充的颜色
                 // 8 位索引或 16 位 RGB 值
            RECT *client); // 客户区域，NULL 表示整个表面
```

用途：

DDraw_Fill_Surface()使用一种颜色来填充表面。该颜色的格式必须与指定表面的色深格式相同，例如，

在 256 色模式下为 1 字节，在高颜色模式下为 RGB 描述符。如果要填充表面的一个子区域，可设置 RECT* 参数，以指定该区域；否则将填充整个表面。如果成功执行，则返回 TRUE。

范例：

```
// 使用颜色值 0 填充主表面
DDraw_Fill_Surface(lpddsprimary,0);
```

函数原型：

```
UCHAR *DDraw_Lock_Surface(LPDIRECTDRAWSURFACE7 lpdds,int *lpitch);
```

用途：

DDraw_Lock_Surface()锁定指定表面（如果可能），并返回指向该表面的指针，同时用该表面的内存跨距更新指定的 lpitch 变量。表面处于锁定状态时，可对它执行操作，将像素写入到其中，但是交换操作将被阻止，因此应该尽可能快地解锁表面。解锁表面后，内存指针和内存跨距很可能无效，所以不应再使用它们。如果成功执行，则返回表面内存的非 NULL 地址，否则返回 NULL 值。

范例：

```
// 用于存储内存跨距
int lpitch = 0;

// 锁定前面创建的 64×64 表面 image
UCHAR *memory = DDraw_Lock_Surface(image, &lpitch);
```

函数原型：

```
int DDraw_Unlock_Surface(LPDIRECTDRAWSURFACE7 lpdds);
```

用途：

DDraw_Unlock_Surface()将解除对表面的锁定，您只需提供表面的指针。如果成功执行，则返回 TRUE。

范例：

```
// 解除对表面 image 的锁定
DDraw_Unlock_Surface(image);
```

函数原型：

```
UCHAR *DDraw_Lock_Back_Surface(void);
UCHAR *DDraw_Lock_Primary_Surface(void);
```

用途：

这两个函数用于锁定主渲染表面和辅助渲染表面。在多数情况下您只会锁定辅助表面，因为这是一个双缓存系统，但是提供锁定主渲染表的功能。如果调用 DDraw_Lock_Primary_Surface()，则下列全局变量将成为有效变量：

```
extern UCHAR *primary_buffer;   // 指向主视频缓存的指针
extern int   primary_lpitch;    // 内存行跨距
```

然后可以根据需求操作表面内存；但是，在锁定期间将阻止交换，且所有硬件加速功能都被禁止。同样，调用 DDraw_Lock_Back_Surface()函数将锁定辅助缓存表面，并使下列变量成为有效变量：

```
extern UCHAR *back_buffer;    // 指向后缓存的指针
extern int   back_lpitch;     // 内存行跨距
```

注意：不要去修改这些全局变量——它们用于跟踪锁定函数中的状态变化。手工修改这些全局变量可能导致引擎崩溃。

范例：

```
// 锁定主缓存表面，并将一个像素写入到左上角
// 这里假设使用的是 8 位颜色模式
DDraw_Lock_Primary( );

primary_buffer[0] = 100;
```

函数原型：

```
int DDraw_Unlock_Primary_Surface(void);
int DDraw_Unlock_Back_Surface(void);
```

用途：

这些函数用于解除对主缓存和辅助缓存表面的锁定。试图解锁一个未被锁定的表面，将不会有任何效果。如果成功执行，则返回 TRUE。

范例：

```
// 解除对辅助缓存的锁定
DDraw_Unlock_Back_Surface( );
```

3.4.8 2D 多边形函数

下面的函数集组成了 2D 多边形系统。它们并不是高级、快速或一流的函数，但是非常适合用于创建演示程序。这些函数管用并能够完成其任务，但还有更好的方法来执行这些任务。

函数原型：

```
void Draw_Triangle_2D(int x1,int y1, // 三角形的顶点
         int x2,int y2,
         int x3,int y3,
         int color, // 8 位颜色索引
         UCHAR *dest_buffer, // 目标缓存
         int mempitch); // 内存跨距

// 定点高速版本，精度稍微低些
void Draw_TriangleFP_2D(int x1,int y1,
            int x2,int y2,
            int x3,int y3,
            int color,
            UCHAR *dest_buffer,
            int mempitch);

// 16 位版本
void Draw_Triangle_2D16(int x1,int y1, // 三角形的顶点
         int x2,int y2,
         int x3,int y3,
         int color, // 16 位 RGB 颜色
         UCHAR *dest_buffer, // 目标缓存
         int mempitch); // 内存跨距
```

用途：

Draw_Triangle_2D*()使用指定颜色在指定内存缓存中绘制一个填充三角形。这个三角形将根据全局变量指定的当前裁剪区域被裁剪，而不是使用 DirectDraw 裁剪区域进行裁剪，因为该函数使用软件（而不是使用硬件）来绘制直线。Draw_TriangleFP_2D()执行相同的操作，但在内部使用定点数学运算，速度快些，但精度差些。这些函数都没有返回值。

范例：

```
// 在后缓存中使用颜色索引 50 对应的颜色绘制下述顶点构成的三角形：
// (100,10)、(150,50)和(50,60)
Draw_Triangle_2D(100,10,150,50,50,60,
         50, // 颜色索引
         back_buffer,
         back_lpitch);

// 使用 16 位版本的函数完成相同的工作
// RGB 颜色为红色
Draw_Triangle_2D16(100,10,150,50,50,60,
         RGB16BIT565(31,0,0), // 假设格式为 5.6.5
         back_buffer,
         back_lpitch);
```

函数原型：

```
inline void Draw_QuadFP_2D(int x0,int y0, // 顶点
```

```
        int x1,int y1,
        int x2,int y2,
        int x3,int y3,
        int color, // 8 位颜色索引
        UCHAR *dest_buffer, // 目标视频缓存
        int mempitch); // 缓存的内存跨距
```

用途：

Draw_QuadFP_2D()以两个三角形的方式绘制指定的四边形，只适用于 8 位颜色模式，没有 16 位颜色版本。该函数没有返回值。

范例：

```
// 绘制一个四边形，顶点必须按顺时针或逆时针方向排列
Draw_QuadFP_2D(0,0, 10,0, 15,20, 5,25,
          100,
          back_buffer, back_lpitch);
```

函数原型：

```
void Draw_Filled_Polygon2D(
     POLYGON2D_PTR poly, // 要渲染的多边形
     UCHAR *vbuffer, // 视频缓存
     int mempitch); // 内存跨距

// 16 位版本
void Draw_Filled_Polygon2D16(
     POLYGON2D_PTR poly, // 要渲染的多边形
     UCHAR *vbuffer, // 视频缓存
     int mempitch); // 内存跨距
```

用途：

Draw_Filled_Polygon2D*()在 8 位颜色模式或 16 位颜色模式下，绘制一个 n 边填充多边形。该函数接受要渲染的多边形、指向视频缓存指针和内存跨距作为其参数。注意该函数相对于多边形的(x0，y0)进行渲染，因此一定要先对它们进行初始化。该函数没有返回值。

范例：

```
// 在 8 位模式下，在主缓存中绘制一个多边形
Draw_Filled_Polygon2D(&poly,
           primary_buffer,
           primary_lpitch);

// 在 16 位模式下，在主缓存中绘制一个多边形
Draw_Filled_Polygon2D16(&poly,
           primary_buffer,
           primary_lpitch);
```

函数原型：

```
int Translate_Polygon2D(
      POLYGON2D_PTR poly, // 要平移的多边形
      int dx, int dy); // 平移因子
```

用途：

Translate_Polygon2D()平移指定多边形的原点(x0，y0)。注意该函数不会平移或修改多边形的顶点。如果成功执行，将返回 TRUE。

范例：

```
// translate polygon 10,-5
Translate_Polygon2D(&poly, 10, -5);
```

函数原型：

```
int Rotate_Polygon2D(
        POLYGON2D_PTR poly, // poly to rotate
        int theta); // angle 0-359
```

用途：

Rotate_Polygon2D() 以顺时针方向绕其原点旋转给定多边形。其中旋转角度必须为一个位于 0～365 之间的整数。如果成功执行，将返回 TRUE。

范例：

```
// rotate polygon 10 degrees
Rotate_Polygon2D(&poly, 10);
```

函数原型：

```
int Scale_Polygon2D(POLYGON2D_PTR poly, // poly to scale
          float sx, float sy); // scale factors
```

用途：

Scale_Polygon2D() 函数分别在 x 轴和 y 轴方向，按比例系统 sx 和 sy 缩放给定多边形。该函数没有返回值。

范例：

```
// scale the poly equally 2x
Scale_Polygon2D(&poly, 2,2);
```

2D 图元处理函数

这套函数包括各种图元处理函数。您应该从来没有看到过这些函数，至少我这么认为。而且，它们也支持 16 位颜色模式。

函数原型：

```
int Draw_Clip_Line(int x0,int y0, // 起点
          int x1, int y1, // 终点
          int color,    // 8 位颜色
          UCHAR *dest_buffer, // 视频缓存
          int lpitch); // 内存跨距

// 16 位版本
int Draw_Clip_Line16(int x0,int y0, // 起点
          int x1, int y1, // 终点
          int color,    // 16 位 RGB 颜色
          UCHAR *dest_buffer, // 视频缓存
          int lpitch); // 内存跨距
```

用途：

Draw_Clip_Line*() 根据当前裁剪矩形对指定直线进行裁剪，然后在指定缓存中绘制裁剪后的直线（8 位或 16 位）。如果成功执行，则返回 TRUE。

范例：

```
// 在后缓存中绘制一条从(10,10)到(100,200)的直线
// 假设使用 8 位颜色模式
Draw_Clip_Line(10,10,100,200,
         5, // 颜色索引
         back_buffer,
         back_lpitch);

// 在后缓存中绘制一条从(10,10)到(100,200)的直线
// 假设使用 16 位颜色模式
Draw_Clip_Line16(10,10,100,200,
         RGB16BIT565(0,31,0), // 绿色
         back_buffer,
         back_lpitch);
```

函数原型：

```
int Clip_Line(int &x1,int &y1,  // 起点
       int &x2, int &y2); // 终点
```

用途：

Clip_Line() 函数大多数时候供内部调用，但您也可以通过调用它来根据当前裁剪矩形对指定的直线进

行裁剪。该函数将修改指定的端点参数，如果您不想要这种效果，应保存这些参数。另外，该函数不绘制任何内容，它只是以数学方式对端点进行裁剪，因此位深无关紧要。如果成功执行，则返回 TRUE。

```
// 对(x1,y1)到(x2,y2)的直线进行裁剪
Clip_Line(x1,y1,x2,y2);
```

函数原型：

```
int Draw_Line(int xo, int yo, // 起点
       int x1,int y1,  // 终点
       int color,    // 8 位颜色索引
       UCHAR *vb_start, // 视频缓存
       int lpitch);   // 内存跨距

// 16 位版本
int Draw_Line16(int xo, int yo, // 起点
       int x1,int y1,  // 终点
       int color,    // 16 位 RGB 颜色
       UCHAR *vb_start, // 视频缓存
       int lpitch);   // 内存跨距
```

用途：

DrawLine*()函数绘制一条直线，不做任何裁剪，因此需要确保端点位于显示表面的有效坐标范围内。该函数执行裁剪操作的版本快些，因为它不执行裁剪操作。如果成功执行，则返回 TRUE。

范例：

```
// 在后缓存中绘制一条从(10,10)到(100,200)的直线
// 8 位颜色模式
Draw_Line(10,10,100,200,
     5, // 颜色索引为 5
     back_buffer,
     back_lpitch);

// 在后缓存中绘制一条从(10,10)到(100,200)的直线
// 16 位颜色模式
Draw_Line16(10,10,100,200,
      RGB16BIT(31,31,31), // 白色
      back_buffer,
      back_lpitch);
```

函数原型：

```
inline int Draw_Pixel(int x, int y, // 像素的位置
           int color, // 8 位颜色
           UCHAR *video_buffer, // 表面
           int lpitch); // 内存跨距

// 16 位版本
inline int Draw_Pixel16(int x, int y, // 像素位置
            int color, // 16 位 RGB 颜色
            UCHAR *video_buffer, // 表面
            int lpitch); // 内存跨距
```

用途：

Draw_Pixel*()在指定显示表面的内存缓存中绘制一个像素。在大多数情况下，您不会基于像素来创建物体，因为多次调用该函数将花费大量的时间。如果不考虑速度，该函数也可以完成相应的工作——至少它是内联的。如果成功执行，则返回 TRUE。

范例：

```
// 使用颜色索引 100 在大小为 640×480×8 的屏幕中央绘制一个像素
Draw_Pixel(320,240, 100, back_buffer, back_lpitch);

// 在大小为 640×480×16 的屏幕中央绘制一个蓝色像素
Draw_Pixel16(320,240, RGB16BIT(0,0,31), back_buffer, back_lpitch);
```

函数原型：

```
int Draw_Rectangle(int x1, int y1, // 左上角
        int x2, int y2, // 右下角
        int color, // 8 位颜色索引或 16 位 RGB 字
        LPDIRECTDRAWSURFACE7 lpdds); // dd 表面
```

用途：

Draw_Rectangle() 在指定 DirectDraw 表面上绘制一个矩形。该表面必须未被锁定，调用该函数才能正常工作。该函数使用硬件加速，因此速度非常快，它可以在 8 位或 16 位颜色模式下工作。如果成功执行，则返回 TRUE。

范例：

```
// 使用黑色填充屏幕
Draw_Rectangle(0,0,639,479,0,lpddsback);
```

函数原型：

```
void HLine(int x1,int x2, // 起点和终点的 x 坐标
      int y,     // 在哪一行绘制
      int color,    // 8 位颜色索引
      UCHAR *vbuffer, // 视频缓存
      int lpitch); // 内存跨距

void HLine16(int x1,int x2, // 起点和终点的 x 坐标
      int y,     // 在哪一行绘制
      int color,    // 16 位 RGB 颜色
      UCHAR *vbuffer, // 视频缓存
      int lpitch); // 内存跨距
```

用途：

HLine*() 以非常快的速度绘制一条水平直线。该函数没有返回值。

范例：

```
// 在 8 位模式下，使用颜色索引 20 绘制一条从(10,100)到(100,100) 的直线
HLine(10,100,100,
   20, back_buffer, back_lpitch);

// 在 16 位模式下，绘制一条从(10,100)到(100,100)的蓝色直线
HLine(10,100,100,
   RGB16Bit(0,0,255), back_buffer, back_lpitch);
```

函数原型：

```
void VLine(int y1,int y2, // 起始行和终止行
      int x,     // 在哪列中绘制
      int color,    // 8 位颜色
      UCHAR *vbuffer,// 视频缓存
      int lpitch);  // 内存跨距

void VLine16(int y1,int y2, // 起始行和终止行
      int x,     // 在哪一列绘制
      int color,    // 16 位 RGB 颜色
      UCHAR *vbuffer,// 视频缓存
      int lpitch);  // 内存跨距
```

用途：

VLine() 以非常快的速度绘制一条垂直直线。它没有 HLine() 函数那么快，但比 Draw_Line() 函数快，如果已知一条直线所有情况下都是垂直的，可使用该函数。该函数没有返回值。

范例：

```
// 在 8 位颜色模式下使用颜色索引 54 绘制一条从(320,0)到(320,479)的直线
VLine(0,479,320,54,
   primary_buffer,
   primary_lpitch);

// 在 16 位颜色模式下使用绿色绘制一条从(320,0)到(320,479)的直线
VLine(0,479,320,RGB16Bit(0,255,0),
   primary_buffer,
```

```
    primary_lpitch);
```

函数原型：

```
void Screen_Transitions(int effect,  // 渐变方式
            UCHAR *vbuffer,// 视频缓存
            int lpitch);  // 内存跨距

// 屏幕渐变命令
#define SCREEN_DARKNESS 0       // 渐变为黑色
#define SCREEN_WHITENESS 1       // 渐变为白色
#define SCREEN_SWIPE_X  2       // 水平扫描
#define SCREEN_SWIPE_Y  3       // 垂直扫描
#define SCREEN_DISOLVE  4       // 像素淡出
#define SCREEN_SCRUNCH  5       // 方块压缩
#define SCREEN_BLUENESS 6       // 渐变为蓝色
#define SCREEN_REDNESS  7       // 渐变为红色
#define SCREEN_GREENNESS 8       // 渐变为绿色
```

用途：

Screen_Transition()函数执行上述头信息中列出的各种内存中屏幕渐变。这些变换是破坏性的，如果变换后需要它们，应保存图像和调色板。该函数只能在 8 位颜色模式下运行。该函数没有返回值。

范例：

```
// 将主显示屏幕渐变为黑色
Screen_Transition(SCREEN_DARKNESS, NULL, 0);
```

函数原型：

```
int Draw_Text_GDI(char *text, // 以 NULL 结尾的字符串
      int x,int y, // 位置
      COLORREF color, // RGB 颜色
      LPDIRECTDRAWSURFACE7 lpdds); // dd 表面

int Draw_Text_GDI(char *text, // 以 NULL 结尾的字符串
        int x,int y, // 位置
        int color, // 8 位颜色索引
        LPDIRECTDRAWSURFACE7 lpdds); // dd 表面
```

用途：

Draw_Text_GDI()将在 8 位或 16 位颜色模式下，使用指定颜色在指定表面的指定位置绘制 GDI 文本。该函数被重载，因此可以接受 RGB()宏形式的 COLORREF 参数或 8 位颜色索引。注意，仅当目标表面未被锁定时该函数才管用，因为它将暂时锁定该表面，以执行 GDI 文本操作。如果成功执行，则返回 TRUE。

范例：

```
// 使用 RGB 颜色(100,100,0)绘制文本
Draw_Text_GDI("This is a test",100,50,
       RGB16Bit(100,100,0),lpddsprimary);
```

3.4.9　数学函数和错误函数

我们将在很大程度上重新编写数学库，但是这里还是介绍它们，因为前面的演示程序可能使用过它们。另外，引擎还提供了错误支持，您可以使用它们在引擎运行时输出错误和诊断文本。这是很有帮助的，因为调试 DirectX 应用程序是非常困难的。

函数原型：

```
int Fast_Distance_2D(int x, int y);
```

用途：

Fast_Distance()使用一种快速近似方法计算从(0，0)到(x，y)的距离。返回距离的误差在 3.5%之内，并被截取为整数。它使用泰勒（Taylor）级数执行近似运算。

范例：

```
int x1=100,y1=200; // 物体 1
```

```
int x2=400,y2=150; // 物体 2

// 计算这两个物体之间的距离
int dist = Fast_Distance_2D(x1-x2, y1-y2);
```

函数原型：

```
float Fast_Distance_3D(float x, float y, float z);
```

用途：

Fast_Distance_3D()使用一种快速近似方法计算从(0，0，0)到(x，y，z)的距离。该函数返回距离的误差在11%之内。

范例：

```
// 计算(0,0,0)和(100,200,300)之间的距离
float dist = Fast_Distance_3D(100,200,300);
```

函数原型：

```
int Find_Bounding_Box_Poly2D(
        POLYGON2D_PTR poly, // 多边形
        float &min_x, float &max_x, // 包围框
        float &min_y, float &max_y);
```

用途：

Find_Bounding_Box_Poly2D()函数用于计算能够包围 poly 指定的多边形的最小矩形。如果成功执行，则返回 TRUE。该函数的参数为引用参数。

范例：

```
POLYGON2D poly; // 假设该变量已经被初始化
int min_x, max_x, min_y, max_y; // 用于存储结果

// 找出包围框
Find_Bounding_Box_Poly2D(&poly,min_x,max_x,min_y,max_y);
```

函数原型：

```
int Open_Error_File(char *filename);
```

用途：

Open_Error_File()打开一个磁盘文件，用于接收通过 Write_Error()函数发送的错误信息。如果成功执行，则返回 TRUE。

范例：

```
// 打开一个错误日志
Open_Error_File("errors.log");
```

函数原型：

```
int Close_Error_File(void);
```

用途：

Close_Error_File()关闭前面打开的错误文件。实际上，它将关闭文件流。如果没有错误文件打开，调用该函数时将不会发生任何事情。如果成功执行，则返回 TRUE。

范例：

```
// 关闭错误系统，它不接受任何参数
Close_Error_File( );
```

函数原型：

```
int Write_Error(char *string, ...); // 错误格式字符串
```

用途：

Write_Error()将一条错误信息写入到前面打开的错误文件中。如果没有打开错误文件，该函数返回 FALSE，且不会发生任何其他事情。注意该函数使用变量参数指示符(...)，因此可以像使用 printf()那样使

用该函数。如果成功执行，则返回 TRUE。

范例：

```
// 发送一些错误消息
Write_Error("\nSystem Starting...");
Write_Error("\nx-vel = %d", y-vel = %d", xvel, yvel);
```

3.4.10　位图函数

下列函数集组成了 BITMAP_IMAGE 和 BITMAP_FILE 操作例程。有加载 8 位、16 位、24 位、32 位位图的函数，也有从位图中提取图像并创建简单的 BITMAP_IMAGE 对象（它们不是 DirectDraw 表面）的函数。另外，还有在 8 位和 16 位颜色模式下绘制这些图像的函数，但没有提供裁剪支持。如果需要裁剪支持，可自己修改源代码，或使用 BOB（Blitter Object，将在本节末尾介绍）。在进行 3D 渲染时，我们可能使用这些函数来帮助加载纹理图和光照图，下面详细介绍它们。

函数原型：

```
int Load_Bitmap_File(BITMAP_FILE_PTR bitmap, // 位图文件指针
          char *filename); // 要从磁盘加载的.BMP 文件
```

用途：

Load_Bitmap_File()从磁盘加载一个.BMP 格式的位图文件到指定的 BITMAP_FILE 结构中，以便能够对其进行处理。该函数加载 8 位、16 位和 24 位的位图以及 8 位.BMP 文件的调色板信息。如果成功执行，则返回 TRUE。

范例：

```
// 从磁盘加载文件 andre.bmp
BITMAP_FILE bitmap_file;

Load_Bitmap_File(&bitmap_file, "andré.bmp");
```

函数原型：

```
int Unload_Bitmap_File(BITMAP_FILE_PTR bitmap); // bitmap to close and unload
```

用途：

Unload_Bitmap_File()函数释放为 BITMAP_FILE 分配的内存。复制图像或完成对位图的操作后，应调用该函数。可以重用该结构，但必须先释放其占用的内存。如果成功执行，则返回 TRUE。

范例：

```
// 关闭前面打开的文件
Unload_Bitmap_File(&bitmap_file);
```

函数原型：

```
int Create_Bitmap(BITMAP_IMAGE_PTR image, // 位图图像
         int x, int y, // 起始位置
         int width, int height, // 位图大小
         int bpp=8); // 位图的位深
```

用途：

Create_Bitmap()在指定位置按指定大小，创建一个 8 位或 16 位的系统内存位图。该位图最初为空，存储在 BITMAP_IMAGE 图像中。该位图不是 DirectDraw 表面，因此不能提供硬件加速功能或裁剪功能。除非将参数 bpp 设置为 16 以创建 16 位位图，否则该函数默认创建 8 位位图，如果成功执行，则返回 TRUE。

注意：BITMAP_FILE 与 BITMAP_IMAGE 之间有很大的差别。BITMAP_FILE 是一个磁盘.BMP 文件，而 BITMAP_IMAGE 是一个系统内存对象，可以移动它或执行其他操作。

范例：

```
// 创建一个 64×64 的 8 位位图图像
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0,0, 64,64);
```

```
// 在(100,100)处创建一个 16 位的 32×32 图像
BITMAP_IMAGE ship2;

// 将参数 bpp 设置为 16 时将创建 16 位的位图图像，否则该函数默认创建 8 位深度
Create_Bitmap(&ship, 0,0, 32,32,16);
```

函数原型：

```
int Destroy_Bitmap(BITMAP_IMAGE_PTR image); // 要删除的位图图像
```

用途：

Destroy_Bitmap()释放创建（8 位或 16 位）BITMAP_IMAGE 对象时分配的内存。使用完对象后，应对其调用该函数——通常在游戏关闭期间或经过一场血战后对象被打死。如果成功执行，则返回 TRUE。

范例：

```
// 删除前面创建的 BITMAP_IMAGE
Destroy_Bitmap(&ship);
```

函数原型：

```
int Load_Image_Bitmap(
 BITMAP_IMAGE_PTR image, // 用于存储图像的位图
 BITMAP_FILE_PTR bitmap, // 要加载的位图文件对象
 int cx,int cy, // 从哪里开始扫描
 int mode); // 图像扫描模式：基于单元格还是绝对坐标

// 16 为版本
int Load_Image_Bitmap16(
 BITMAP_IMAGE_PTR image, // 用于存储图像的位图
 BITMAP_FILE_PTR bitmap, // 要加载的位图文件对象
 int cx,int cy, // 从哪里开始扫描
 int mode); // 图像扫描模式：基于单元格还是绝对坐标

#define BITMAP_EXTRACT_MODE_CELL 0
#define BITMAP_EXTRACT_MODE_ABS  1
```

用途：

Load_Image_Bitmap*()从 BITMAP_FILE 对象中扫描一幅图像到指定的 BITMAP_IMAGE 存储区域中。当然，位图文件和位图对象必须具相同的位深（8 位或 16 位），且必须使用合适版本的函数。

要使用该函数，首先必须加载一个 BITMAP_FILE，并创建一个 BITMAP_IMAGE 对象。然后调用该函数，从存储在 BITMAP_FILE 中的位图数据中扫描出一幅图像。该函数有两种工作模式：单元格模式和绝对模式。

在单元模式下（BITMAP_EXTRACT_MODE_CELL），扫描图像时假设所有图像都存储在给定大小的模板.BMP 文件中，单个格之间有宽度为 1 像素的边界，单元的大小通常为 8×8、16×16、32×32、64×64 等。在附带光盘上本章对应目录中，包含大量的模板——TEMPLATE*.BMP。单元格按着从左到右、从上到下的顺序编号，并从(0，0)开始。

第二种模式是绝对坐标模式（BITMAP_EXTRACT_MODE_ABS）。在这种模式下，将根据参数 cx 和 cy 指定的坐标来扫描图像。要从相同.BMP 文件中加载不同大小的图像时，可以使用这种模式。因此，您不能对它们进行模板化。

范例：

```
// 8 位范例
// 假设源.BMP 位图文件是 640×480×8，且是一个 8×8 的单个格阵列
// 每个单元格的大小为 32×32
// 要加载第 2 行中从左边数起的第 3 个单元格，可以像下面这样做

// 将.BMP 文件加载到内存中
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file,"images.bmp");
```

```
// 初始化位图
BITMAP_IMAGE ship;
Create_Bitmap(&ship, 0,0, 32,32);

// 扫描数据
Load_Image_Bitmap(&ship, &bitmap_file, 2,1,
        BITMAP_EXTRACT_MODE_CELL);

// 16 位范例
// 假设源.BMP 位图文件是 640×480×16，且是一个 8×8 的单个格阵列
// 每个单元格的大小为 32x32
// 要加载第 2 行中从左边数起的第 3 个单元格，可以像下面这样做

// 将.BMP 文件加载到内存中
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file,"images.bmp");

// 初始化位图
BITMAP_IMAGE ship2;
Create_Bitmap(&ship2, 0,0, 32,32,16);

// 扫描数据
Load_Image_Bitmap16(&ship2, &bitmap_file, 2,1,
        BITMAP_EXTRACT_MODE_CELL);
```

要使用绝对模式加载相同的图像（假设它仍在模板中），必须计算坐标——别忘了图像的周围有宽度为 1 像素的边框。

```
// 8 位范例
Load_Image_Bitmap(&ship, &bitmap_file,
        2*(32+1)+1,1*(32+1)+1,
        BITMAP_EXTRACT_MODE_ABS);
```

函数原型：

```
int Draw_Bitmap(BITMAP_IMAGE_PTR source_bitmap, // 要绘制的位图
        UCHAR *dest_buffer, // 视频缓存
        int lpitch, // 内存跨距
        int transparent); // 是否透明?

// 16 位版本
int Draw_Bitmap16(BITMAP_IMAGE_PTR source_bitmap, // 要绘制的位图
        UCHAR *dest_buffer, // 视频缓存
        int lpitch, // 内存跨距
        int transparent); // 是否透明?
```

用途：

Draw_Bitmap*()在 8 位或 16 位颜色模式下，以透明或不透明方式将指定位图绘制到目标内存表面上。如果参数 transparent 为 1，则启用透明方式，不复制颜色索引为 0 或 RGB 为 0.0.0 的像素。如果成功执行，则返回 TRUE。

范例：

```
// 在后缓存中绘制前面的位图 ship
// 8 位模式下
Draw_Bitmap( &ship, back_buffer, back_lpitch, 1);

// 16 位模式下
Draw_Bitmap16( &ship2, back_buffer, back_lpitch, 1);
```

函数原型：

```
int Flip_Bitmap(UCHAR *image, // 要垂直翻转的图像位
        int bytes_per_line, // 每行多少字节
        int height); // 总行数（高度）
```

用途：

Flip_Bitmap()通常被内部调用，用于在加载期间将.BMP 文件上下倒置，但您有时也可能想使用它来倒置图像（8 位或 16 位）。该函数在内存中执行操作，逐行反转位图，所以原始数据将被反转，使用时一

定要注意！如果成功执行，则返回 TRUE。

范例：

```
// 翻转前面的小型图像 ship
// 假设为 8 位模式，每个像素 1 字节
Flip_Bitmap(ship->buffer, ship->width, ship->height);

// 在 16 位模式下，调用的函数相同
// 但必须将宽度乘以 2，因为每个像素 2 字节
Flip_Bitmap(ship2->buffer, 2*ship2->width, ship2->height);
```

函数原型：

```
int Copy_Bitmap(BITMAP_IMAGE_PTR dest_bitmap,
        int dest_x, int dest_y,
        BITMAP_IMAGE_PTR source_bitmap,
        int source_x, int source_y,
        int width, int height);
```

用途：

Copy_Bitmap()函数从一个位图复制一个矩形区域到另一个位图中。源位图和目标位图可以相同，但源区域和目标区域不能重叠，否则结果将不确定。如果成功执行，则返回 TRUE。

范例：

```
// 将.BMP 文件加载到内存中
BITMAP_FILE bitmap_file;
Load_Bitmap_File(&bitmap_file,"playfield.bmp");


// 初始化一个位图，用于存储 playfield
BITMAP_IMAGE playfield;
Create_Bitmap(&playfield, 0,0, 400,400,16);
// 扫描数据
Load_Image_Bitmap16(&playfield, &bitmap_file, 0,0,
        BITMAP_EXTRACT_MODE_ABSOLUTE);


// 将位图的上半部分复制到下半部分
Copy_Bitmap(&playfield, 0,200,
      &playfield, 0,0,
      200,200);
```

函数原型：

```
int Scroll_Bitmap(BITMAP_IMAGE_PTR image, int dx, int dy=0);
```

用途：

Scroll_Bitmap()将指定位图水平滚动 dx 和垂直滚动 dy。参数为正时将向右/下滚动，参数为负时向左/上滚动。如果成功执行，则返回 TRUE。

范例：

```
// 将 playfield 向右滚动 2 个像素
Scroll_Bitmap(&playfield, 2,0);
```

3.4.11 8 位调色板函数

下列函数组成了 8 位调色板接口。仅当显示模式为 8 位时这些函数才有用。在窗口显示模式中，前 10 种颜色和后 10 种颜色供 Windows 使用，您不能修改它们；当然，在全屏模式下可以修改它们。

另外，用于存储颜色的基本数据结构是 Win32 结构 PALETTEENTRY：

```
typedef struct tagPALETTEENTRY {
  BYTE peRed;  // 红色分量（8 位）
  BYTE peGreen;  // 绿色分量（8 位）
  BYTE peBlue;   // 蓝色分量（8 位）
  BYTE peFlags;  // 标记，Windows 颜色为 PC_EXPLICIT，其他颜色为 PC_NOCOLLAPSE
```

```
} PALETTEENTRY;
```

下面来看一下各个调色板函数。

函数原型：

```
int Set_Palette_Entry(
        int color_index, // 要修改其颜色的索引
        LPPALETTEENTRY color); // 颜色
```

用途：

Set_Palette_Entry()函数修改调色板中的一种颜色。您只需提供一个 0～255 的颜色索引和一个包含颜色的 PALETTEENTRY 指针，在下一帧中更新将生效。另外，该函数还将更新阴影调色板。该函数的运行速度很慢，如果要更新整个调色板，应使用 Set_Palette()函数。如果成功执行，则返回 TRUE；否则返回 FALSE。

范例：

```
// 将索引 0 对应的颜色设置为黑色
PALETTEENTRY black = {0,0,0,PC_NOCOLLAPSE};
Set_Palette_Entry(0,&black);
```

函数原型：

```
int Get_Palette_Entry(
       int color_index, // 要检索的颜色索引
       LPPALETTEENTRY color); // 用于存储颜色
```

用途：

Get_Palette_Entry()从当前调色板中取回一个调色板项，速度非常快，因为它从基于 RAM 的阴影调色板中获取数据。因此，可以尽情地调用该函数，而不会影响硬件的性能。然而，如果使用函数 Set_Palette_Entry()或 Set_Paltte()来修改系统调色板，阴影调色板将不会相应地更新，从中获取的数据可能是无效数据。如果成功执行，返回 TRUE；否则返回 FALSE。

范例：

```
// 读取调色板项 100
PALETTEENTRY color;
Get_Palette_Entry(100,&color);
```

函数原型：

```
int Save_Palette_To_File(
        char *filename, // 要将调色板保存其中的文件的名称
        LPPALETTEENTRY palette); // 要保存的调色板
```

用途：

Save_Palette_To_File()函数将指定调色板数据保存到磁盘上的 ASCII 文件中，供以后检索或处理。如果动态地生成了一个调色板并想将它保存到磁盘上，该函数将非常方便。然而，该函数假设调色板中的指针指向一个 256 项的调色板，使用时一定要小心。如果成功执行，返回 TRUE；否则返回 FALSE。

范例：

```
PALETTEENTRY my_palette[256]; // 假设该调色板已经创建好

// 保存上述调色板
// 文件名无关紧要，但作者喜欢使用*.pal
Save_Palette_To_File("/palettes/custom1.pal",&my_palette);
```

函数原型：

```
int Load_Palette_From_File(
     char *filename, // 要从中加载数据的文件
     LPPALETTEENTRY palette); // 调色板
```

用途：

Load_Palette_From_File()函数从磁盘加载以前通过 Save_Palette_To_File()函数保存的 256 色调色板。您只需提供文件名和用于存储调色板的数据结构，该函数将把调色板从磁盘加载到数据结构中。然而，该

函数不会将调色板项加载到硬件调色板中，您必须自己通过 Set_Palette() 函数手工完成这种加载。如果成功执行，返回 TRUE；否则返回 FALSE。

范例：

```
// 加载前面保存的调色板
Load_Palette_From_File("/palettes/custom1.pal",&my_palette);
```

注意：通过调用 DDRWA_INI( )函数将显示模式设置为 256 色时，该函数将加载一个标准调色板，它覆盖了最优的颜色空间。该调色板文件名为 PALDATA2.DAT。它是一个包含 256 行的 ASCII 文件，每行都由 0－255 的 R、G、B 值组成。

函数原型：

```
int Set_Palette(LPPALETTEENTRY set_palette); // 要加载到硬件中的调色板
```

用途：

Set_Palette() 函数将指定调色板数据加载到硬件调色板中，同时更新阴影调色板。如果成功执行，返回 TRUE；否则返回 FALSE。

范例：

```
// 将调色板加载到硬件中
Set_Palette(&my_palette);
```

函数原型：

```
int Save_Palette(LPPALETTEENTRY sav_palette); // 调色板变量
```

用途：

Save_Palette() 扫描硬件调色板，将其保存到 sav_palette 中，以便能够将它保存到磁盘中或对其进行处理。save_palette 必须有足够的存储空间来存储 256 种颜色。

范例：

```
// 读取当前的 DirectDraw 硬件调色板
PALETTEENTRY hardware_palette[256];
Save_Palette(&hardware_palette);
```

函数原型：

```
int Rotate_Colors(int start_index, // 起始索引（0～255）
        int end_index); // 终止索引（0～255）
```

用途：

Rotate_Colors() 以循环方式旋转颜色项。它直接操纵硬件调色板。如果成功执行，返回 TRUE；否则返回 FALSE。

范例：

```
// 旋转整个调色板
Rotate_Colors(0,255);
```

函数原型：

```
int Blink_Colors(int command, // 闪光灯引擎命令
        BLINKER_PTR new_light, // 闪光灯数据
        int id); // 闪光灯 ID
```

用途：

Blink_Colors() 函数用于创建异步调色板动画。该函数所涉及的内容很多，这里无法详细介绍。

3.4.12 实用函数

函数原型：

```
DWORD Get_Clock(void);
```

用途：

Get_Clock() 函数返回 Windows 启动后过去的时间，单位为毫秒。

范例：

```
// 读取 Windows 启动后过去的时间
DWORD start_time = Get_Clock( );
```

函数原型：

```
DWORD Start_Clock(void);
```

用途：

Start_Clock() 函数调用 Get_Clock() 函数，并将时间存储在一个全局变量中。然后您可以调用 Wait_Clock()，它在您调用 Start_Clock() 后等待指定的毫秒数。该函数返回调用时的时钟值。

范例：

```
// 开始计时并设置全局变量
Start_Clock( );
```

函数原型：

```
DWORD Wait_Clock(DWORD count); // 等待多长时间，单位为毫秒
```

用途：

Wait_Clock() 函数在调用 Start_Clock() 后等待指定的毫秒数。该函数返回调用时的时钟计数。然而，在指定时间到达之前，该函数不会返回。

范例：

```
// 等待 30 毫秒
Start_Clock( );

// 其他代码

Wait_Clock(30);
```

注意：本书后面将使用 Windows 支持的性能更高的计时器，以提高时间精度。

函数原型：

```
int Collision_Test(int x1, int y1, // 物体 1 的左上角坐标
          int w1, int h1, // 物体 1 的宽度和高度
          int x2, int y2, // 物体 2 的左上角坐标
          int w2, int h2); // 物体 2 的宽度和高度
```

用途：

Collisioin_Test() 函数检测指定的两个矩形是否重叠。您必须提供每个矩形的左上角坐标以及宽度和高度。如果成功执行，返回 TRUE；否则返回 FALSE。

范例：

```
// 检测两个 BITMAP_IMAGE 是否重叠
if (Collision_Test(ship1->x,ship1->y,ship1->width,ship1->height,
          ship2->x,ship2->y,ship2->width,ship2->height))
  { // hit

  } // end if
```

函数原型：

```
int Color_Scan(int x1, int y1, // 矩形的左上角坐标
        int x2, int y2, // 矩形的右下角坐标
        UCHAR scan_start, // 起始扫描颜色
        UCHAR scan_end, // 终止扫描颜色
        UCHAR *scan_buffer, // 要扫描的内存
        int scan_lpitch); // 内存跨距
```

用途：

Color_Scan() 函数是另一种碰撞检测算法，它扫描一个矩形，以检测一个 8 位值或某个范围内的一系列值。可以使用它来判断某个颜色索引是否出现在某个区域中。当然，它只适用于 8 位图像，但其源代码很容易扩展到 16 位或更高的颜色模式。如果发现指定颜色，则返回 TRUE。

范例：

```
// 扫描范围[122, 124]中的颜色索引
Color_Scan(10,10, 50, 50, 122,124, back_buffer, back_lpitch);
```

3.4.13 BOB（Blitter 对象）引擎

虽然通过少量编程，可以用 BITMAP_IMAGE 类型做您想要的任何事情，但它不是最好的方式——它没有使用 DirectDraw 表面，所以不支持硬件加速。因此，我创建了一种叫作 BOB（Blitter 对象）的类型，它非常类似于子画面。子画面（sprite）是一个可以在屏幕上移动而不会影响屏幕背景的对象。在我们这里，情况并不是这样，所以我将动画对象称为 BOB，而不是子画面。

本书很少使用 BOB 引擎，但是我仍想介绍它，因为它是使用 DirectDraw 表面和 2D 加速的典范。下面简要介绍一下 BOB，首先是 BOB 的数据结构：

```
// BOB
typedef struct BOB_TYP
    {
    int state;      // 状态
    int anim_state;  // 动作状态变量
    int attr;       // 属性
    float x,y;       // 位置
    float xv,yv;      // 速度
    int width, height;  // 宽度和高度
    int width_fill;    // 用于强制表面的宽度为 8 的倍数
    int bpp;        // 每个像素占用的位数
    int counter_1;    // 计数器
    int counter_2;
    int max_count_1;  // 阈值
    int max_count_2;
    int varsI[16];    // 深度为 16 的整数栈
    float varsF[16];   // 深度为 16 的浮点数栈
    int curr_frame;    // 当前动画帧
    int num_frames;    // 总动画帧数
    int curr_animation; // 当前动作索引
    int anim_counter;   // 用于给动作切换定时
    int anim_index;    // 动作元素索引
    int anim_count_max; // 几帧后再切换动作
    int *animations[MAX_BOB_ANIMATIONS]; // 动作
    // 位图图像 DD 表面
    LPDIRECTDRAWSURFACE7 images[MAX_BOB_FRAMES];
    } BOB, *BOB_PTR;
```

BOB 是由一个或多个 DirectDraw 表面（当前最多 64 个）表示的图形对象。可以移动 BOB、绘制 BOB、使用 BOB 制作动画、使之处于运动状态。BOB 考虑当前 DirectDraw 裁剪矩形，因此将被裁剪和加速。图 3.10 说明了 BOB 及其与其动画帧的关系。

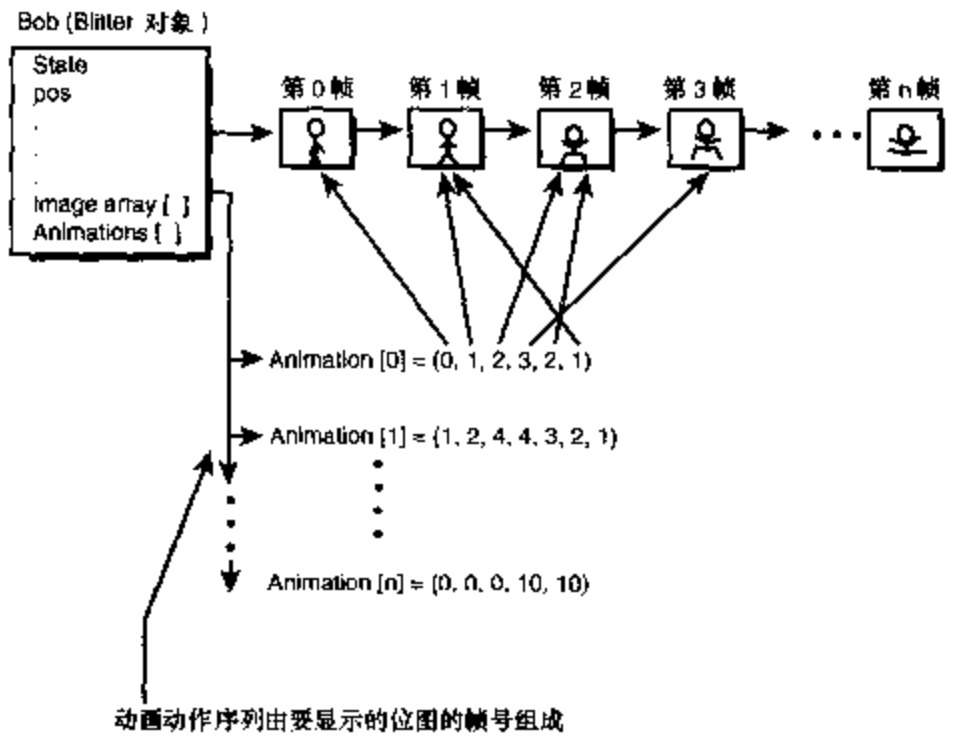


图 3.10 BOB 动画系统

BOB 引擎支持 8 位或 16 位图像和动画序列，您可以加载一组帧和一个动画序列，然后该序列将可以播放帧中的内容。这是一种非常棒的功能。另外，BOB 能够在 8 位和 16 位颜色模式下工作，大多数函数有其内部逻辑，能够检测到系统当前的位深，因此您不用考虑这些问题。惟一需要显式地为 16 位颜色模式调用不同版本的函数是 Load_Frame_BOB*()（加载 BOB）和 Draw_BOB*()（绘制 BOB），本章最后将介绍它们，并提供一些 8/16 位演示范例。

最后，所有 BOB 函数在成功执行时都返回 TRUE，否则返回 FALSE。下面逐一介绍这些函数。

函数原型：

```
int Create_BOB(BOB_PTR bob, // 指向要创建的 BOB 的指针
  int x, int y,      // BOB 的初始位置
  int width, int height,  // BOB 的大小
  int num_frames, // 总帧数
  int attr,    // 属性
  int mem_flags=0, // 表面内存标记，0 表示 VRAM
  USHORT color_key_value=0, // 颜色键值
                // 根据 bbp 的设置被解释为 8 位索引或 16 位 RGB 值
  int bpp=8); // bits per pixel, default is 8
```

用途：

Create_BOB()创建一个 BOB 对象，并对它进行设置。该函数分配所有内部变量，并为每帧创建一个单独的 DirectDraw 表面。大多数变量的含义都是不言自明的，惟一需要解释的一个参数是属性变量 attr。关于每个属性的详细说明，参见表 3.1，可以使用逻辑 OR 将这些属性组合起来，用于设置参数 attr。

表 3.1　　有效的 BOB 属性

值	含义
BOB_ATTR_SINGLE_FRAME	创建只有一帧的 BOB
BOB_ATTR_MULTI_FRAME	创建一个有多帧的 BOB，但 BOB 动画是一个由第 0~n 帧组成的线性序列
BOB_ATTR_MULTI_ANIM	创建一个支持动画序列的多帧 BOB
BOB_ATTR_ANIM_ONE_SHOT	如果设置了该标记，则播放动画序列时只播放一次，然后停止。这时，内部变量 anim_state 将被设置。要再次播放动画序列，需要重置该变量
BOB ATTR_BOUNCE	该标记使 BOB 像球一样，在屏幕边界弹回，仅当使用 Move_BOB( )时才有效
BOB_ATTR_WRAPAROUND	该标记使 BOB 移动到屏幕边界时回绕到屏幕的另一侧，仅当使用 Move_BOB( )时才有效

范例：

下面是一些创建 BOB 的范例。首先，在(50，100)处创建一个大小为 96×64 的 8 位单帧 BOB：

```
BOB car;

// 创建 BOB
if (!Create_BOB(&car, 50,100, // BOB 的地址和初始位置
        96,64,    // BOB 的大小
        1,       // 帧数
        BOB_ATTR_SINGLE_FRAME, // 属性
        0, // 内存标记
        0, // 颜色键值，在 8 位模式下为颜色索引
        8)) // 每个像素的位数， 在 256 色模式下为 8 位
  { /* 错误处理代码 */ }
```

下面的代码创建一个大小为 32×32 像素，包含 8 帧的 16 位 BOB：

```
BOB ship;

// 创建 BOB
if (!Create_BOB(&ship, 0,0, // BOB 的地址和初始位置
        32,32, // BOB 的大小
        8,   // 帧数
        BOB_ATTR_MULTI_FRAME, //属性
        0, // 内存标记
```

```
        0, // 颜色键值，在 16 位模式下为 RGB 字
        16)) // 每个像素的位数，在 16 位模式下为 16
  { /* 错误处理代码 */ }
```

最后，创建一个支持动画序列的 8 位多帧 BOB：

```
BOB greeny;

// 创建 BOB
if (!Create_BOB(&greeny, 0,0,
  32,32,32,BOB_ATTR_MULTI_ANIM,0,0,8))
  { /* 错误处理代码 */ }
```

提示：Create_BOB( )函数的最后 3 个参数都有默认值。因此，如果不想覆盖它们，可不必提供这些参数。

函数原型：

```
int Destroy_BOB(BOB_PTR bob); // 指向要删除的 BOB 的指针
```

用途：

Destroy_BOB()将删除以前创建的 BOB。BOB 是 8 位还是 16 位无关紧要。

范例：

```
// 删除前面创建的 BOB
Destroy_BOB(&greeny);
```

函数原型：

```
int Draw_BOB(BOB_PTR bob, // 指向要绘制的 BOB 的指针
 LPDIRECTDRAWSURFACE7 dest); // 目标表面

// 16 位版本
int Draw_BOB16(BOB_PTR bob, // 指向要绘制的 BOB 的指针
 LPDIRECTDRAWSURFACE7 dest); // 目标表面
```

用途：

Draw_BOB*()是一个功能非常强大的函数，它在指定的 DirectDraw 表面上绘制指定的 BOB。将在当前位置绘制 BOB 中的当前帧（动画参数指定）。另外，应该根据 BOB 是 8 位还是 16 位的，选择适当版本的函数；否则，有可能只得到半个 BOB！

警告：要让该函数能正常工作，目标表面必须未被锁定。

范例：

```
// 将一个多帧 BOB 放到后缓存(50,50)处，然后绘制该 BOB 的第 1 帧
BOB ship;

// 创建 BOB，默认为 8 位
if (!Create_BOB(&ship, 0,0,
        32,32,8,BOB_ATTR_MULTI_FRAME,0))

// 加载 BOB 图像，稍后再介绍
// 设置位置和当前帧
ship.x = 50;
ship.y = 50;
ship.curr_frame = 0; // 要绘制的帧

// 绘制 BOB
Draw_BOB(&ship, lpddsback);
```

函数原型：

```
int Draw_Scaled_BOB(BOB_PTR bob, // 指向要绘制的 BOB 的指针
 int swidth, int sheight, // 新的 BOB 宽度和高度
 LPDIRECTDRAWSURFACE7 dest); // 目标表面

// 16 位版本
int Draw_Scaled_BOB16(BOB_PTR bob, // 指向要绘制的 BOB 的指针
 int swidth, int sheight, // 新的 BOB 宽度和高度
```

```
 LPDIRECTDRAWSURFACE7 dest); // 目标表面
```

用途：

Draw_Scaled_BOB*()函数与 Draw_BOB()函数非常类似，但可以指定绘制出的 BOB 的宽度和高度，这样 BOB 将相应地被缩放。这种功能非常有用，如果有硬件加速功能，这将是一种缩放 BOB 以获得 3D 效果的有效方法。实际上，我们将使用一种类似的技术为 3D 应用程序创建广告牌。

注意：广告牌是以 3D 方式绘制的 2D 图像，但总是与观察方向垂直。例如，在 DOOM 中，所有生物都是广告牌子画面，而游戏世界是基于多边形的。

范例：

```
// 以大小为 128×128 的方式绘制 BOB ship
// 虽然其原来的大小为 32×32 像素
// 8 位版本
Draw_Scaled_BOB(&ship, 128,128,lpddsback);
```

函数原型：

```
int Load_Frame_BOB(
 BOB_PTR bob, // 指向要将帧加载到其中的 BOB 的指针
 BITMAP_FILE_PTR bitmap,// 指向要从中扫描数据的位图的指针
 int frame,  // 将图像加载到哪一帧中
 int cx,int cy, // 用单元格或坐标表示的扫描位置
 int mode);   // 扫描模式，与 Load_Frame_Bitmap( )中相同

// 16 位版本
int Load_Frame_BOB16(
 BOB_PTR bob, // 指向要将帧加载到其中的 BOB 的指针
 BITMAP_FILE_PTR bitmap, // 指向要从中扫描数据的位图的指针
 int frame,  // 将图像加载到哪一帧中
 int cx,int cy, // 用单元格或坐标表示的扫描位置
 int mode);   // 扫描模式，与 Load_Frame_Bitmap( )中相同
```

用途：

Load_Frame_BOB*()函数的功能与 Load_Frame_Bitmap()完全相同，有关细节请参阅该函数。惟一的区别是，新增了指定将图像加载到哪一帧中的功能。如果您创建了一个有 4 帧的 BOB，将一帧一帧地加载它们。另外，必须根据数据和屏幕的位深，使用正确版本的函数（8 位或 16 位）。

范例：

```
// 这是一个以单元格模式从一个位图文件中加载 4 帧到一个 16 位 BOB 中的例子

BOB ship; // BOB
// 将位图文件 bitmap16bit 中的单元格(0,0)、(1,0)、(2,0)和 3,0)
// 加载到 BOB 的第 0、1、2、3 帧中

for (int index=0; index<4; index++)
   Load_Frame_BOB16(&ship,&bitmap16bit,
           index, index,0,
           BITMAP_EXTRACT_MODE_CELL );
```

注意：下面的 BOB 操作函数集对 BOB 的数据结构元素进行操作，因此对于 8 位和 16 位的 BOB，它们的功能完全相同。

函数原型：

```
int Load_Animation_BOB(
  BOB_PTR bob,  // bob to load animation into
  int anim_index, // which animation to load 0..15
  int num_frames, // number of frames of animation
  int *sequence); // 指向存储动画序列的数组的指针
```

用途：

Load_Animation()函数有必要解释一下。该函数用于设置 BOB 内部的 16 个数组之一，这些数组包含了动画序列，如图 3.10 所示。

范例：

假设有一个 BOB，它包含 8 帧，编号分别是 0、1、...7，且定义了如下 4 个动作：

```
int anim_walk[] = {0,1,2,1,0};
int anim_fire[] = {5,6,0};
int anim_die[]  = {3,4};
int anim_sleep[] = {0,0,7,0,0};
```

要设置 BOB 的这些动作，可以这样做：

```
// 创建一个多动作 BOB
if (!Create_BOB(&alien, 0,0, 32,32,8,BOB_ATTR_MULTI_ANIM,0))
  { /* 错误处理代码 */ }

// 加载 BOB 帧的代码...
// 使用数组 anim_walk 来设置动作 0
Load_Animation_BOB(&alien, 0,5,anim_walk);

// 使用数组 anim_fire 来设置动作 1
Load_Animation_BOB(&alien, 1,3,anim_fire);

// 使用数组 anim_die 来设置动作 2
Load_Animation_BOB(&alien, 2,2,anim_die);

// 使用数组 anim_sleep 来设置动作 3
Load_Animation_BOB(&alien, 3,5,anim_sleep);
```

设置动作后，可以指定活动动作，并使用后面的函数来播放它。

函数原型：

```
int Set_Pos_BOB(BOB_PTR bob, // 指向要设置其位置的 BOB 的指针
       int x, int y); // 新位置
```

用途：

Set_Pos_BOB()函数是一种设置 BOB 位置的简单方法。它除了对内部变量(*x*, *y*)赋值外没做任何其他事情。

范例：

```
// 设置 BOB alien 的位置
Set_Pos_BOB(&alien, player_x, player_y);
```

函数原型：

```
int Set_Vel_BOB(BOB_PTR bob, // 指向要设置其速度的 BOB 的指针
     int xv, int yv); // 新速度
```

用途：

每个 BOB 都一个由(*xv*, *yv*)指定的速度。Set_Vel_BOB()函数根据传入的参数修改这两个值。仅当您使用函数 Move_BOB()移动 BOB 时，BOB 中的速度值才有意义。然而，也可以使用变量 xv 和 yv 来跟踪 BOB 的速度。

范例：

```
// 让 BOB 沿一条水平线移动
Set_Vel_BOB(&alien, 10,0);
```

函数原型：

```
int Set_Anim_Speed_BOB(BOB_PTR bob, // BOB 指针
         int speed); // 动画速度
```

用途：

Set_Anim_Speed()将 BOB 的内部动画速度设置为 anim_count_max。这个数值越大，动画速度越慢；这个数值越小（最小为 0），动画速度越快。然而，仅当您使用内部 BOB 函数 Animate_BOB()时才有意义。当然，您必须创建了一个包含多帧的 BOB。

范例：

```
// 将动画速度设置为每帧连续播放 30 次
Set_Anim_Speed_BOB(&alien, 30);
```

函数原型：

```
int Set_Animation_BOB(
     BOB_PTR bob, // 指向要设置其当前动作的 BOB 的指针
     int anim_index); // 动作索引
```

用途：

Set_Animation_BOB()函数设置 BOB 将播放的动作。在前面的 Load_Animation_BOB()范例中，我们创建了 4 个动作。

范例：

```
// 将编号为 2 的动作设置为活动状态
Set_Animation_BOB(&alien, 2);
```

注意：该函数还会将当前帧设置为当前动作中的第 1 帧。

函数原型：

```
int Animate_BOB(BOB_PTR bob); // 执行 BOB 的指针
```

用途：

Animate_BOB()函数使用 BOB 来生成动画。通常，您将在每帧中调用该函数一次，以更新 BOB 动画。

范例：

```
Animate_BOB(&alien);
```

函数原型：

```
int Move_BOB(BOB_PTR bob); // 指向要移动的 BOB 的指针
```

用途：

Move_BOB()函数根据 Set_Vel_BOB()函数设置的(xv，yv)来移动 BOB，根据属性，BOB 可能从屏幕边界反弹回来、环回到屏幕另一侧或仅仅向前移动。与 Animate_BOB()函数类似，您可以在主循环中调用 Animate_BOB()函数之后（或之前）调用该函数一次。

范例：

```
// 播放 BOB 中的下一帧
Animate_BOB(&alien);

// 移动 BOB
Move_BOB(&alien);
```

函数原型：

```
int Hide_BOB(BOB_PTR bob); // 指向要隐藏的 BOB 的指针
```

用途：

Hide_BOB()函数将 BOB 的内部可见性标记设置为 0，之后 Draw_BOB()将不再显示该 BOB。

范例：

```
// 隐藏 BOB
Hide_BOB(&alien);
```

函数原型：

```
int Show_BOB(BOB_PTR bob); // 指向要显示的 BOB 的指针
```

用途：

Show_BOB()函数将 BOB 的可见性标记设置为 1，使之被绘制出来(相当于撤销 Hide_BOB()函数调用)。下面是一个先隐藏，然后显示 BOB 的范例，因为您要显示一个 GDI 对象或其他东西，不想被 BOB 挡住。

范例：

```
// Hide_BOB(&alien);
// 调用 Draw_BOB 和 GDI 函数等
Show_BOB(&alien);
```

函数原型：

```
int Collision_BOBS(BOB_PTR bob1, // 指向第一个 BOB 的指针
          BOB_PTR bob2); // 指向第二个 BOB 的指针
```

用途：

Collision_BOBS()函数检测两个 BOB 的包围矩形是否重叠，可用于在游戏中进行碰撞检测，以判断玩家 BOB 是否被导弹 BOB 或其他东西击中。

范例：

```
// 检测导弹 BOB 是否击中了玩家 BOB
if (Collision_BOBS(&missile, &player))
  { /* 播放爆炸声 */ }
```

有关 T3DLIB 游戏库的图形模块（T3DLIB1.CPP|H）就介绍到这里，下面介绍 T3DLIB 的输入模块。

3.5 T3DLIB2 DirectX 输入系统

编写一套封装 DirectInput 的简单封装很容易，只需创建一个 API，它包含一个非常简单的接口和几个参数。该接口至少应该支持下列功能：

- 初始化 DirectInput 系统；
- 设置并读取键盘、鼠标和游戏杆；
- 从任何输入设备读取数据；
- 关闭和释放所有资源。

我创建了一个这样的 API，您可以从附带光盘上的 T3DLIB2.CPP|H 中找到。该 API 执行初始化 DirectInput 和读取任何设备所需的一切操作。介绍这些函数之前，先来看一下图 3.11，它描述了每种设备和数据流之间的关系。

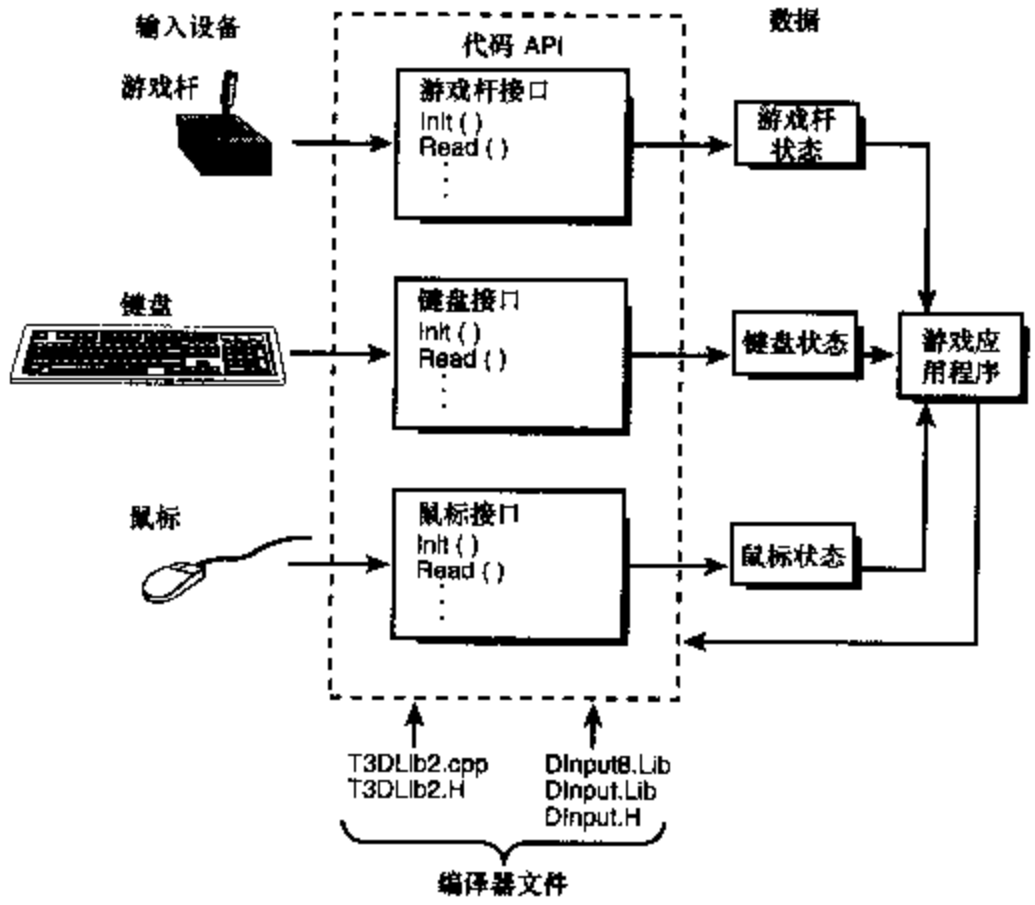


图 3.11 DirectInput 软件系统

下面是这个库中的全局变量：

```
LPDIRECTINPUT8     lpdi;     // dinput 对象
LPDIRECTINPUTDEVICE8 lpdikey;  // dinput 键盘
LPDIRECTINPUTDEVICE8 lpdimouse; // dinput 鼠标
```

```
LPDIRECTINPUTDEVICE8 lpdijoy;  // dinput 游戏杆
GUID  joystickGUID; // 主游戏杆的 guid
char  joyname[80]; // 游戏杆的名称

// 所有输入都存储在这些记录中
UCHAR keyboard_state[256]; // 键盘状态表
DIMOUSESTATE mouse_state; // 鼠标状态
DIJOYSTATE joy_state;   // 游戏杆状态
int joystick_found;    // 指出游戏杆是否插入
```

来自键盘的输入存储在 keyboard_state[ ]中，鼠标数据存放到 mouse_state 中，游戏杆数据存储在 joy_state 中。这些记录的结构都是标准 DirectInput 设备状态结构（键盘数据除外，它使用一个布尔型 BYTES 数组），如下所示：

```
// 鼠标数据
typedef struct DIMOUSESTATE {
  LONG lX; // x 坐标
  LONG lY; // y 坐标
  LONG lZ; // z 坐标
  BYTE rgbButtons[4]; // 鼠标按钮的状态
} DIMOUSESTATE, *LPDIMOUSESTATE;

// 游戏杆数据
typedef struct DIJOYSTATE {
  LONG  lX; // x 坐标
  LONG  lY; // y 坐标
  LONG  lZ; // z 坐标
  LONG  lRx; // 绕 x 轴旋转角度
  LONG  lRy; // 绕 y 轴旋转角度
  LONG  lRz; // 绕 z 轴旋转角度
  LONG  rglSlider[2]; // u、v 滑块（slider）的位置
  DWORD  rgdwPOV[4];  // 视点状态
  BYTE  rgbButtons[32]; // 按钮 0～31 的状态
} DIJOYSTATE, *LPDIJOYSTATE;
```

一般而言，鼠标和游戏杆的(x，y)位置的含义相同：也就是说，您将通过字段 lX 和 lY 来访问它们，而按钮状态是存储在 rgbButtons[ ]中的布尔值。下面介绍函数。变量 joystick_found 是一个布尔值，在您请求访问游戏杆时被设置。如果发现游戏杆，将被设置为 TRUE，否则为 FALSE。使用这个变量，可以条件性地决定是否执行使用游戏杆的代码块。下面介绍这个新的 API：

函数原型：

```
int DInput_Init(void);
```

用途：

Dinput_Init()函数初始化 DirectInput 输入系统。它创建主 COM 对象，如果成功则返回 TRUE，否则返回 FALSE。同时，全局变量 lpdi 将成为有效变量。但是，该函数不创建任何设备。下面是一个初始化输入系统的范例：

范例：

```
if (!DInput_Init( ))
  { /* 错误处理代码 */ }
```

函数原型：

```
void DInput_Shutdown(void);
```

用途：

Dinput_Shutdown()函数释放所有 COM 对象以及调用 DInput_Init()函数期间分配的所有资源。通常，应该在释放所有输入设备后调用 Dinput_Shutdown()函数。下面是一个关于输入系统的范例：

范例：

```
DInput_Shutdown( );
```

函数原型：

```
DInput_Init_Keyboard(void);
```

用途：

DInput_Init_Keyboard()函数初始化键盘并开始获取键盘输入。它应该总是能够正常工作并返回 TRUE，除非另一个 DirectX 应用程序以不合作方式接管了键盘。下面是一个范例：

范例：

```
if (!DInput_Init_Keyboard( ))
  { /*错误处理代码 */ }
```

函数原型：

```
int DInput_Init_Mouse(void);
```

用途：

DInput_Init_Mouse()函数初始化鼠标并开始获取鼠标输入。该函数不带参数，如果成功则返回 TRUE，否则返回 FALSE。它应该总是能够正常工作，除非没有插入鼠标或另一个 DirectX 应用程序完全接管了鼠标。如果一切正常，lpdimouse 将成为有效的接口指针。下面是一个范例：

范例：

```
if (!DInput_Init_Mouse( )) { /* 错误处理代码 */ }
```

函数原型：

```
int DInput_Init_Joystick(int min_x=-256, // 最小 x 值
        int max_x=256, // 最大 x 值
        int min_y=-256, // 最小 y 值
        int max_y=256, // 最大 y 值
        int dead_zone=10); // 死区（dead zone）比例
```

用途：

DInput_Init_Joystick()初始化游戏杆设备。该函数接受 5 个参数，分别定义了从游戏杆发送回来的 x、y 范围以及死区比例。如果要使用默认值-256 到 256 以及 10%的死区，可以不提供参数，因为它们有默认值。如果该函数返回 TRUE，表示已经发现游戏杆并对其进行了设置和初始化。调用该函数后，接口指针 lpdijoy 将有效。此外，字符串 joyname[ ]将包含游戏杆设备的“友好”名称，如“Microsoft Sidewinder Pro”等。

下面是一个范例，它初始化游戏杆并设置其 x、y 范围为-1024 到 1024，死区为 5%。

范例：

```
if (!DInput_Init_Joystick(-1024, 1024, -1024, 1024, 5))
  { /* 错误处理代码 */ }
```

函数原型：

```
void DInput_Release_Joystick(void);
void DInput_Release_Mouse(void);
void DInput_Release_Keyboard(void);
```

用途：

DInput_Release_Joystick()、DInput_Release_Mouse()和 DInput_Release_Keyboard()分别释放相应的输入设备。即使没有初始化这些设备，也可以调用这些函数，所以您尽可以在应用程序结束时调用它们。下面是一个完整的范例，它首先初始化 DirectInput 系统，初始化所有设备，然后释放输入设备并关闭输入系统：

范例：

```
// 初始化 DirectInput 系统
DInput_Init( );

// 初始化所有输入设备并接管它们
DInput_Init_Joystick( );
DInput_Init_Mouse( );
DInput_Init_Keyboard( );

// 输入循环代码...

// 首先释放所有设备，顺序无关紧要
DInput_Release_Joystick( );
```

```
DInput_Release_Mouse( );
DInput_Release_Keyboard( );

// 关闭 DirectInput
DInput_Shutdown( );
```

函数原型：

```
int DInput_Read_Keyboard(void);
```

用途：

DInput_Read_Keyboard()扫描键盘状态，并将数据存储到数组 keyboard_state[ ]中，后者是一个包含 256 个元素的 BYTE 数组。这是标准 DirectInput 键盘状态数组，要使它的意义很明确，必须使用 DirectInput 键盘常量 DIK_*。键被按下时，相应元素的值为 0x80。下面是一个范例，它使用 DirectInput 键盘常量（有关这些常量，可查询 SDK 或表 3.2）来判断左箭头键和右箭头键是否被按下。

范例：

```
// 读取键盘
if (!DInput_Read_Keyboard( ))
  /* 错误处理代码 */ }

// 检测状态数据
if (keyboard_state[DIK_RIGHT])
  { /* 将飞船右移 */ }
else
if (keyboard_state[DIK_LEFT])
  { /* 将飞船左移 */ }
```

表 3.2　　DirectInput 键盘状态常量

符号	含义	符号	含义
DIK_ESCAPE	Esc 键	DIK_NUMPAD0-9	数字键盘上的 0～9
DIK_0-9	主键盘上的 0～9	DIK_ADD	数字键盘上的加号
DIK_MINUS	减号键	DIK_NUMPADENTER	数字键盘上的回车键
DIK_EQUALS	等号键	DIK_RCONTROL	右 Ctrl 键
DIK_BACK	退格键	DIK_RMENU	右 Alt 键
DIK_TAB	制表键	DIK_HOME	方向键盘上的 Home 键
DIK_A-Z	字母 A～Z	DIK_UP	方向键盘上的向上箭头键
DIK_LBRACKET	左括号	DIK_PRIOR	方向键盘上的 PgUp
DIK_RBRACKET	右括号	DIK_LEFT	方向键盘上的向左箭头键
DIK_RETURN	主键盘上的回车键	DIK_RIGHT	方向键盘上的向右箭头键
DIK_LCONTROL	左 Ctrl 键	DIK_END	方向键盘上的 End 键
DIK_LSHIFT	左 Shift 键	DIK_DOWN	方向键盘上的向下箭头键
DIK_RSHIFT	右 Shift 键	DIK_NEXT	方向键盘上的 PgDn 键
DIK_LMENU	左 Alt 键	DIK_INSERT	方向键盘上的 Insert 键
DIK_SPACE	空格键	DIK_DELETE	方向键盘上的 Delete 键
DIK_F1-15	功能键 F1～F15		

函数原型：

```
int DInput_Read_Mouse(void);
```

用途：

DInput_Read_Mouse()读取相对鼠标状态并将结果存储在 mouse_state 中，后者是一个 DIMOUSESTATE 结构。数据是相对 delta 模式。大多数情况下，只需查看 mouse_state.1X、mouse_state.1Y 和 rgbButtons[0..2]（针对 3 个鼠标按钮的布尔值）。下面的范例读取鼠标状态并据此移动光标和绘制图像。

范例：

```
// 读取鼠标
if (!DInput_Read_Mouse( ))
  { /* 错误处理代码 */ }
// 移动光标
cx+=mouse_state.lX;
cy+=mouse_state.lY;

// 检测左按钮是否被按下
if (mouse_state.rgbButtons[0])
  Draw_Pixel(cx,cy,col,buffer,pitch);
```

函数原型：

```
int DInput_Read_Joystick(void);
```

用途：

DInput_Read_Joystick()轮询游戏杆，然后将数据读取到joy_state中，后者是一个DIJOYSTATE结构。当然，如果没有游戏杆，该函数将返回FALSE且joy_state无效。如果成功，joy_state将包含游戏杆的状态信息。返回的数据将位于前面设置的范围内，而按钮值是存储在 rgbButtons[ ]中的布尔值。下面的范例演示了如何使用游戏杆向左或向右移动飞船，并按下第一个按钮来开火。

范例：

```
// 读取游戏杆数据
if (!DInput_Read_Joystick( ))
  { /* 错误处理代码 */ }

// 移动飞船
ship_x+=joy_state.lX;
ship_y+=joy_state.lY;

// 检测是否开火
if (joy_state.rgbButtons[0])
  { // 开火 // }
```

当然，游戏杆可能有很多按钮和多个轴。在这种情况下，可以使用 joy_state 的其他字段，这是在DIJOYSTATE DirectInput结构中定义的。

注意：虽然我们使用了接口 IDIRECTINPUTDEVICE8，但不需要使用 DJOYSTATE2 结构。它只适用于力反馈（force feedback）设备。

接下来介绍大多数游戏程序员最不喜欢处理的一个子系统——声音和音乐！然而，在 DirectX 下，并使用我编写的封装函数，产生声音效果和播放音乐就像玩一样简单。

3.6 T3DLIB3 声音和音乐库

我使用了《Windows 游戏编程大师技巧》中介绍的所有声音和音乐技术来创建游戏引擎的最后一个组件——T3DLIB3。它由两个主要的源代码文件组成：

- T3DLIB3.CPP——C/C++源代码；
- T3DLIB3.H——头文件。

然而，还需包括 DirectSound 导入库 DSOUND.LIB 以进行链接。DirectMusic 没有导入库，它是一个纯粹的 COM 对象，所以没有 DMUSIC.LIB。另一方面，仍需要将 DirectSound 和 DirectMusic 的.H 文件的位置告诉编译器，以便编译期间能够找到它们。这些头文件是：

- DSOUND.H——标准 DirectSound 头文件。
- DMKSCTRL.H、DMUSICI.H、DMUSICC.H、DMUSICF.H——这些都是用于 DirectMusic 的。

下面来看一看 T3DLIB3.H 头文件的主要元素。

注意：在 DirectX 8.0+中，微软公司将 DirectSound 和 DirectMusic 集成到一起，并称之为 DirectAudio。

我没有看出这样做有何意义，本书将分别使用这两个组件。

3.6.1　头文件

头文件 T3DLIB3.H 包含了 T3DLIB3.CPP 使用的类型、宏和外部变量。下面是这个头文件中的#define 语句：

```
#define DM_NUM_SEGMENTS 64 // 可存储到内存中 midi 节（segment）数

// midi 对象状态常量
#define MIDI_NULL    0  // 未加载
#define MIDI_LOADED  1  // 已加载
#define MIDI_PLAYING 2  // 已加载且正在播放
#define MIDI_STOPPED 3  // 已加载，但没有播放

#define MAX_SOUNDS   256 // 系统可同时加载的最大声音数

// 数字声音对象状态常量
#define SOUND_NULL   0 // " "
#define SOUND_LOADED  1
#define SOUND_PLAYING 2
#define SOUND_STOPPED 3
```

该文件中的宏不多，有一个用于将 0～100 转换为微软分贝数，还有一个用于将多字节字符转换为宽字符。

```
#define DSVOLUME_TO_DB(volume) ((DWORD)(-30*(100 - volume)))

// 从多字节格式转换为 Unicode
#define MULTI_TO_WIDE( x,y )
    MultiByteToWideChar( CP_ACP,MB_PRECOMPOSED, y,-1,x,_MAX_PATH)
```

警告：在本书中打印时，宏代码可能被分成两行；在实际程序中，宏必须位于同一行中。

下面介绍用于声音引擎的类型，首先是 DirectSound 对象。

3.6.2　类型

声音引擎只有两种类型：一种用于存储数字样本，另一个用于存储 MIDI 节：

```
// 用于存储单个声音
typedef struct pcm_sound_typ
  {
  LPDIRECTSOUNDBUFFER dsbuffer; // directsound 缓冲区
                  // 包含声音
  nt state;  // 状态
  int rate;  // 播放速度
  int size;  // 大小
  int id;   // id 号
  } pcm_sound, *pcm_sound_ptr;
```

下面是 DirectMusic 节类型：

```
// directmusic MIDI 节
typedef struct DMUSIC_MIDI_TYP
{
IDirectMusicSegment    *dm_segment; // directmusic 节
IDirectMusicSegmentState  *dm_segstate; // 状态
int             id;      // ID
int             state;   // MIDI 歌曲的状态

} DMUSIC_MIDI, *DMUSIC_MIDI_PTR;
```

声音和 MIDI 节由引擎分别存储到上述两种结构中。下面介绍全局变量。

3.6.3　全局变量

T3DLIB3 中包含很多全局变量。首先介绍 DirectSound 系统的全局变量：

```
LPDIRECTSOUND  lpds;  // directsound 接口指针
```

```
DSBUFFERDESC  dsbd;  // directsound 描述
DSCAPS     dscaps; // directsound caps
HRESULT    dsresult // directsound 结果
DSBCAPS    dsbcaps; // directsound 缓冲区 caps

pcm_sound  sound_fx[MAX_SOUNDS]; // 声音缓冲区数组
WAVEFORMATEX  pcmwf; // 通用波形格式结构
```

下面是 DirectMusic 的全局变量：

```
// directmusic 全局变量
IDirectMusicPerformance *dm_perf ;  // directmusic 性能管理器
IDirectMusicLoader     *dm_loader; // directmusic 加载函数

// 用于存储所有的 directmusic midi 对象
DMUSIC_MIDI     dm_midi[DM_NUM_SEGMENTS];
int dm_active_id;  //当前处于活动状态的 midi 节
```

注意：其中以粗体显示的是用于存储声音和 MIDI 节的数组。

对于这些全局变量，您没有必要了解，除非要直接访问接口。一般而言，API 将为您处理一切事情。这个库分为两部分：DirectSound 和 DirectMusic。我们先来看一下 DirectSound。

3.6.4 DirectSound API 封装函数

DirectSound 可能很复杂，也可能很简单，这取决于您如何使用它。如果想要一个“全能”API，可能需要使用大多数的 DirectSound 函数；但如果您想要一个比较简单的 API，它能够初始化 DirectSound、加载并播放特定格式的声音，则只需要封装几个函数，相对来说容易得多。

我做的工作是，将设置 DirectSound 的工作组织成一套函数。另外，我对声音系统进行了一个抽象，让您使用一个在加载过程中提供的 ID 来引用声音（对于 DirectMusic 也是这样），然后使用该 ID 来播放声音、检查其状态或结束播放。这样，您将无需关心任何复杂的接口指针。这个新的 API 支持下列功能：

- 调用函数来初始化和关闭 DirectSound；
- 加载 11KHz、8 位单声道格式的.WAV 文件；
- 播放已加载的声音文件；
- 停止播放；
- 检测声音的播放状态；
- 改变音量、播放速度或立体声左右声道的相对强度（stereo panning）；
- 从内存中删除声音。

下面逐一介绍这些函数。

注意：除非特别声明，所有函数在执行成功后都返回 TRUE（1），否则返回 FALSE（0）。

函数原型：

```
int DSound_Init(void);
```

用途：

Dsound_Init()函数用于初始化整个 DirectSound 系统。它创建 DirectSound COM 对象，设置优先级别等。要使用声音，应该在应用程序开始时调用该函数。下面是一个范例：

范例：

```
if (!DSound_Init(void))
  { /* 错误处理代码 */ }
```

函数原型：

```
int DSound_Shutdown(void);
```

用途：

Dsound_Shutdown()函数用于关闭 DirectSound 系统并释放在调用 Dsound_Init()函数期间创建的所有

COM 接口。然而，Dsound_Shutdown()不会释放分配给声音的内存，您必须自己手动调用另一个函数来完成该任务。下面是一个用于关闭 DirectSound 系统的范例：

范例：

```
if (!DSound_Shutdown( ))
  { /* 错误处理代码 */ }
```

函数原型：

```
int DSound_Load_WAV(char *filename);
```

用途：

Dsound_Load_WAV()函数将创建一个 DirectSound 缓冲区，加载声音数据文件到内存中，并为播放该声音做准备。该函数接受要加载文件的完整路径和文件名（包括扩展名.WAV）作为其参数，并从磁盘中加载该文件。如果执行成功，返回一个非负 ID 号。您必须保存该数字，因为它将被用作引用该声音的句柄；如果找不到指定的声音文件，或加载了太多的声音，将返回-1。下面的范例加载一个名为 FIRE.WAV 的声音文件。

范例：

```
int fire_id = DSound_Load_WAV("FIRE.WAV");

// test for error
if (fire_id==-1)
  { /* 错误处理代码 */}
```

当然，如何保存该 ID 由您决定，可考虑使用一个数组或其他数据结构。

最后，您可能想知道声音数据存放在哪里以及如何处理它。如果您真正需要，可以访问 pcm_sound 数组 sound_fx[ ]内的数据，并使用从加载函数获得的 ID 作为索引。例如，下面的范例使用 ID sound_id 访问声音的 DirectSound 缓冲区。

范例：

```
sound_fx[sound_id].dsbuffer
```

函数原型：

```
int DSound_Replicate_Sound(int source_id); // 要复制的声音的 ID
```

用途：

Dsound_Replicate_Sound()函数复制一个声音，而不复制用于存储声音的内存。例如，假设有一个开枪射击的声音，并要连续开枪射击三次；完成这项任务的惟一方法是复制三份射击声音拷贝到三个不同的 DirectSound 内存缓冲区中，但这种方法将浪费一些内存。

这里有一种解决方案，除实际声音数据外，可以创建一份声音缓冲区的副本。但是不是复制声音数据，而是使用一个指针指向该声音，DirectSound 将自动识别声音数据源。如果要播放枪击声音 8 次，应加载该声音一次，生成 7 份副本，并获得 8 个惟一的 ID。复制的声音与正常声音一样，但您不是使用 Dsound_Load_WAV()函数来加载并创建声音，而是使用 Dsound_Replicate_Sound()函数复制声音。下面的范例创建 8 次枪击声音。

范例：

```
int gunshot_ids[8]; // 用于存储声音 ID

// 加在主声音
gunshot_ids[0] = Load_WAV("GUNSHOT.WAV");

// 制作副本
for (int index=1; index<8; index++)
  gunshot_ids[index] = DSound_Replicate_Sound(gunshot_ids[0]);

// 随意使用 gunshot_ids[0..7]
```

函数原型：

```
int DSound_Play_Sound(int id,    // 要播放的声音的 ID
          int flags=0, // 0 或 DSBPLAY_LOOPING
          int volume=0, // 未用
```

```
            int rate=0,  // 未用
            int pan=0);  // 未用
```

用途：

Dsound_Play_Sound() 函数播放以前加载的声音。您只需提供声音的 ID 和播放标记（0 表示播放一次，DSBPLAY_LOOPING 表示循环播放），然后声音将开始播放。如果声音正在播放，它将从头开始播放。下面的范例加载并播放声音。

范例：

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
DSound_Play_Sound(fire_id,0);
```

可以省略标记 0，因为该参数的默认为 0：

```
int fire_id = DSound_Load_WAV("FIRE.WAV");
DSound_Play_Sound(fire_id);
```

不管采用哪种方式，FIRE.WAV 声音都将播放一次，然后停止。要循环播放，应将标记参数设置为 DSBPLAY_LOOPING。

函数原型：

```
int DSound_Stop_Sound(int id);
int DSound_Stop_All_Sounds(void);
```

用途：

Dsound_Stop_Sound() 函数停止正在播放的声音。您只需提供声音的 ID。Dsound_Stop_All_Sounds() 函数停止当前所有正在播放的声音。下面的范例停止播放声音 fire_id。

范例：

```
DSound_Stop_Sound(fire_id);
```

在程序结束时，应在退出前停止播放所有声音。可以对每个声音分别调用 Dsound_Stop_Sound() 函数，也可以调用 Dsound_Stop_All_Sounds() 函数停止播放所有声音。

```
//系统关闭代码...
DSound_Stop_All_Sounds( );
```

函数原型：

```
int DSound_Delete_Sound(int id); // 要删除的声音的 ID
int DSound_Delete_All_Sounds(void);
```

用途：

Dsound_Delete_Sound() 函数从内存中删除一个声音，并释放与它相关的 DirectSound 缓冲区。如果该声音正在播放，则先停止它。Dsound_Delete_All_Sounds() 函数删除所有以前加载的声音。下面的范例删除声音 fire_id。

范例：

```
DSound_Delete_Sound(fire_id);
```

函数原型：

```
int DSound_Status_Sound(int id);
```

用途：

DSound_Status_Sound() 函数根据 ID 检测一个已加载声音的状态。您只需将声音的 ID 号提供给该函数，它将返回下列值之一：

- DSBSTATUS_LOOPING——声音当前正在播放且处于循环模式。
- DSBSTATUS_PLAYING——声音当前正在播放且处于单次播放模式。

如果 Dsound_Status_Sound() 函数返回的值不是上述值之一，表明声音没有处于播放状态。下面是一个完整的范例，它等待一个声音播放完毕后删除它。

范例：

```
// 初始化 DirectSound
DSound_DSound_Init( );

// 加载声音
int fire_id = DSound_Load_WAV("FIRE.WAV");

// 以单次模式播放声音
DSound_Play_Sound(fire_id);

// 等待声音播放完毕
while(DSound_Sound_Status(fire_id) &
      (DSBSTATUS_LOOPING | DSBSTATUS_PLAYING));

// 删除声音
DSound_Delete_Sound(fire_id);
// 关闭 DirectSound
DSound_DSound_Shutdown( );
```

函数原型：

```
int DSound_Set_Sound_Volume(int id,  // 声音的 ID
          int vol); // 音量的取值范围为 0～100
```

用途：

Dsound_Set_Sound_Volume()函数实时地修改声音的音量。您只需提供声音的 ID 和一个 0～100 的值，然后音量将立即改变。下面的范例将一个声音的音量变为加载时的 50%。

范例：

```
DSound_Set_Sound_Volume(fire_id, 50);
```

可以使用下面的语句将音量调整为 100%：

```
DSound_Set_Sound_Volume(fire_id, 100);
```

函数原型：

```
int DSound_Set_Sound_Freq(
        int id,  // 声音 ID
        int freq); // 新的播放速度，取值范围为 0-100000
```

用途：

Dsound_Set_Sound_Freq()函数修改声音的播放速度。由于所有的声音都必须以 11KHz 单声道加载，因此下面的范例将声音的播放速度加快一倍。

范例：

```
DSound_Set_Sound_Freq(fire_id, 22050);
```

要使声音像 Darth Vader，可使用下面的语句：

```
DSound_Set_Sound_Freq(fire_id, 6000);
```

函数原型：

```
int DSound_Set_Sound_Pan(
  int id,  // 声音 ID
  int pan); // 相对强度，取值范围为-10000 到 10000
```

用途：

Dsound_Set_Sound_Pan()函数设置声音在左右两个扬声器的相对强度。-10000 表示纯粹在左边，10000 表示声音纯粹在右边。要让两边的强度相同，可将 pan 参数设置为 0。下面的范例将声音设置为右声道。

范例：

```
DSound_Set_Sound_Pan(fire_id, 10000);
```

3.6.5　DirectMusic API 封装函数

DirectMusic API 比 DirectSound API 要简单些。我已经创建了一些函数，用于初始化 DirectMusic、创

建所有 COM 对象，让您可以将注意力集中在如何加载和播放 MIDI 文件上。下面是其基本功能列表：

- 调用函数来初始化和关闭 DirectMusic；
- 从磁盘加载 MIDI 文件；
- 播放 MIDI 文件；
- 停止正在播放的 MIDI；
- 检测 MIDI 节的播放状态；
- 如果 DirectSound 已被初始化，则自动连接到 DirectSound；
- 从内存中删除 MIDI 节。

下面逐个介绍这些函数。

注意：除非特别声明，所有函数在执行成功时都返回 TRUE（1），否则返回 FALSE（0）。

函数原型：

```
int DMusic_Init(void);
```

用途：

DMusic_Init() 函数初始化 DirectMusic，并创建所有必要的 COM 对象。应在调用其他任何 DirectMusic 库函数之前调用该函数。此外，要使用 DirectSound，应在调用 DMusic_Init() 函数前初始化 DirectSound。下面是一个使用该函数的范例.

范例：

```
if (!DMusic_Init( ))
  { /* 错误处理代码 */ }
```

函数原型：

```
int DMusic_Shutdown(void);
```

用途：

DMusic_Shutdown() 函数关闭整个 DirectMusic 引擎。它释放所有 COM 对象，卸载所有已加载的 MIDI 节。应该在应用程序结束时调用该函数，如果使用了 DirectSound，该调用还应位于关闭 DirectSound 的调用之前。下面是一个范例。

范例：

```
if (!DMusic_Shutdown( ))
  { /* 错误处理代码 */ }

// 现在关闭 DirectSound
```

函数原型：

```
int DMusic_Load_MIDI(char *filename);
```

用途：

DMusic_Load_MIDI() 函数加载一个 MIDI 节到内存中，并分配 midi_ids[ ]数组中的一条记录。该函数返回被加载的 MIDI 节的 ID，如果执行不成功则返回-1。返回的 ID 将用作对 MIDI 节的引用，供其他函数使用。下面的范例加载两个 MIDI 文件。

范例：

```
// 加载文件
int explode_id = DMusic_Load_MIDI("explosion.mid");
int weapon_id = DMusic_Load_MIDI("laser.mid");

// 检测文件
if (explode_id == -1 || weapon_id == -1)
  { /* 出现了问题*/ }
```

函数原型：

```
int DMusic_Delete_MIDI(int id);
```

用途：

DMusic_Delete_MIDI()函数从系统中删除以前加载的 MIDI 节。您只需提供要删除的 MIDI 节的 ID。下面的范例从内存中删除前一个范例中加载的 MIDI 文件。

范例：

```
if (!DMusic_Delete_MIDI(explode_id) ||
  !DMusic_Delete_MIDI(weapon_id) )
{ /* 错误处理代码 */ }
```

函数原型：

```
int DMusic_Delete_All_MIDI(void);
```

用途：

DMusic_Delete_All_MIDI()函数从系统中删除所有 MIDI 节。下面是一个范例。

范例：

```
// 删除前面加载的两个 MIDI 节
if (!DMusic_Delete_All_MIDI( ))
  { /* 错误处理代码 */ }
```

函数原型：

```
int DMusic_Play(int id);
```

用途：

DMusic_Play()从头开始播放 MIDI 节。您只需提供播放的 MIDI 节的 ID。下面是一个范例。

范例：

```
// 加载文件
int explode_id = DMusic_Load_MIDI("explosion.mid");

// 播放
if (!DMusic_Play(explode_id))
  { /* 错误处理代码 */ }
```

函数原型：

```
int DMusic_Stop(int id);
```

用途：

DMusic_Stop()函数停止当前正在播放的 MIDI 节。如果该 MIDI 节已停止，该函数不会有任何效果。下面是一个范例。

范例：

```
// 停止播放激光冲击波声音
if (!DMusic_Stop(weapon_id))
  { /* 错误处理代码 */ }
```

函数原型：

```
int DMusic_Status_MIDI(int id);
```

用途：

DMusic_Status()函数根据 ID 检测 MIDI 节的状态。状态编码如下：

```
#define MIDI_NULL   0  // 未加载
#define MIDI_LOADED  1  // 已加载
#define MIDI_PLAYING 2  // 已加载且正在播放
#define MIDI_STOPPED 3  // 已加载但没有播放
```

下面的范例根据一个 MIDI 节是否播放完毕来修改游戏的状态。

范例：

```
// 主游戏循环
while(1)
   {
```

```
if (DMusic_Status(explode_id) == MIDI_STOPPED)
 game_state = GAME_MUSIC_OVER;

} // end while
```

有关 DirectSound 和 DirectMusic API 就介绍到这里。本章后面将提供一个使用 DirectSound 和 DirectMusic API 的演示程序，现在我们继续构建最终的 T3D 游戏控制台和虚拟计算机接口。

T3DLIB 库一瞥

至此，我们有了组成 T3D 库的 3 个主要.CPP|H 模块：

- T3DLIB1.CPP|H——DirectDraw 和图形算法；
- T3DLIB2.CPP|H——DirectInput；
- T3DLIB3.CPP|H——DirectSound 和 DirectMusic。

编译演示程序时，应查看其包含的.H 文件，如果其中包括上述任何一个.H 文件，则还应包含相应的.CPP 文件。

编译任何程序时，都应确保已经包括了所有的库源文件和头文件以及 DirectX .LIB 文件，同时将编译器设置为输出 Win32 .EXE 目标文件。

3.7 建立最终的 T3D 游戏控制台

至此，我们有足够的功能来实现虚拟图形接口，包括声音支持和输入支持。另外，虚拟接口模型可以利用《Windows 游戏编程大师技巧》开发的 T3DLIB 库模块中的数据结构、全局变量和函数来实现。我们的目标是实现一个双缓存图形系统，它支持 8/16 位窗口模式和全屏模式，且提供了虚拟计算机接口功能。

接下来需要以本章前面的 alpha 版本为基础，创建一个真正的游戏控制台“模板”。但在此之前，先来看一看虚拟图形接口的功能以及到真实图形接口的映射。声音和音乐方面的内容很容易理解，因此这里不介绍它们的映射。本章后面几节中的演示程序使用了很多声音、音乐和输入，您可以在真实程序中看到这些功能。下面来看一看图形部分。

3.7.1 映射真实图形到虚拟接口的非真实图形

本章前面设计了一个最通用的图形接口，可以将它视为在任何计算机上实现基于软件的 3D 图形学的最低需求。现在，有一些库作为工具后，来看一看从虚拟到现实的最终映射。

1. 启动系统

我们假设在虚拟计算机的图形接口中，有一个名为 Create_Window() 的函数，其原型如下：

```
Create_Window(int width, int height, int bit_depth);
```

它将负责处理图形系统准备就绪的所有细节，并根据指定的大小和位深打开一个窗口。在我们的真实实现中，这项功能由两个函数调用完成，因为我们想将 Windows 与 DirectX 分开。第一个函数调用是标准的 Windows 调用，它创建一个标准的 Windows 窗口。

请注意，要创建全屏应用程序，应使用窗口标记 WM_POPUP；要创建窗口应用程序，应使用其他窗口标记，如 WM_OVERLAPPEDWINDOW。因此，将图形系统准备就绪是一个由两步组成的过程：

第 1 步，使用适当的参数调用 Win32 API 函数，创建一个全屏窗口或窗口应用程序。对于全屏窗口，通常不需要任何控件。

第 2 步，使用窗口句柄（它是一个全局变量）和适当的参数，调用 DirectX 封装函数完成准备工作。

所有操作实际上都是在第 2 步完成的。在这一步骤中，将初始化 DirectDraw、创建帧缓存、为 8 位颜色模式生成调色板，并将裁剪矩形同图形窗口和后缓存关联起来。

由于我们将使用游戏控制台模板，因此先在 WinMain() 函数中打开窗口，然后在 Game_Init() 函数中调

用 DDraw_Init()，完成初始化工作。读者可能还记的，该函数的原型如下：

```
int DDraw_Init(int width, int height, int bpp, int windowed=0);
```

2. 全局映射

我们在虚拟图形接口中设计的第一项内容是两个帧缓存：一个可见缓存，一个离屏缓存。我们将它们称为主缓存和辅助（后）缓存，如图 3.12 所示。

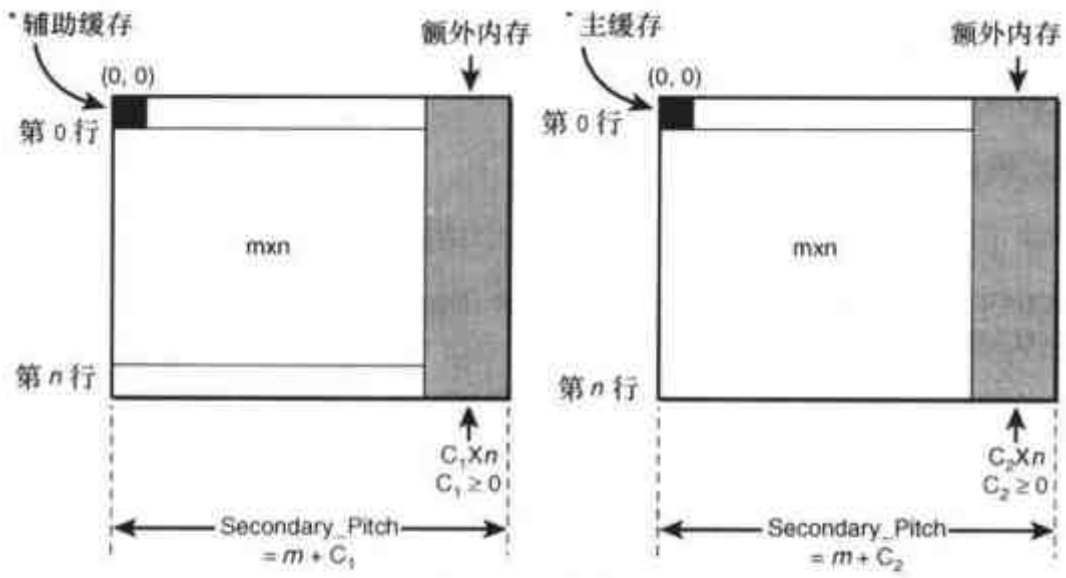


图 3.12　帧缓存

另外，对于任何给定的分辨率和位深，这些缓存都是可线性寻址的，但有一个限制，即行与行之间的内存跨距可能与像素跨距不同，因此需要使用变量来记录它。虚拟帧缓存指针和内存跨距变量的名称如下：

```
UCHAR *primary_buffer; // 主缓存
int  primary_pitch; // 内存跨距，单位为字节

UCHAR *secondary_buffer; // 辅助缓存
int  seconday_pitch;  // 内存跨距，单位为字节
```

在实际图形库 T3DLIB1.CPP 中，用于上述目的的变量如下：

```
LPDIRECTDRAWSURFACE7 lpddsprimary;   // dd 主表面
LPDIRECTDRAWSURFACE7 lpddsback;    // dd 后表面

UCHAR        *primary_buffer; // 主视频缓存
UCHAR        *back_buffer;   // 辅助缓存
int         primary_lpitch;  // 内存行跨距
int         back_lpitch;   // 内存行跨距
```

您可能注意到了，还有两个面向 DirectX 的变量：lpddsprimary 和 lpddsback。它们分别是指向主表面和辅助表面的 DirectDraw 表面指针。有些调用需要它们，您应该了解它们。另外，主缓存和辅助缓存的宽度和高度总是相同的。虽然主缓存的客户区域可能只是一个窗口（而不是整个桌面），但辅助缓存的大小总是与客户区域相同。即可见的主帧缓存和不可见的辅助帧缓存的大小和位深总是相同。

3. 256 色模式

虽然在 T3DLIB1.CPP|H 的图形 API 清单中，您已经见过了 256 色调色板操作函数，但这里还想重复一下，让您能够操纵调色板的函数如下：

```
int Set_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Get_Palette_Entry(int color_index, LPPALETTEENTRY color);
int Load_Palette_From_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette_To_File(char *filename, LPPALETTEENTRY palette);
int Save_Palette(LPPALETTEENTRY sav_palette);
int Set_Palette(LPPALETTEENTRY set_palette);
```

其中以粗体显示的函数最重要，它们用于修改单个调色板项或一次性修改整个调色板。在大多数情况下，每次修改一个调色板项的效率很低，最好一次性修改整个调色板。您传递的数据结构为指向 PALETTENTRY 的指针或是指向 PALETTENTRY 数组的指针。下面是数据结构 PALETTENTRY 的定义：

```
typedef struct tagPALETTEENTRY {
  BYTE peRed;  // 红色分量（8 位）
  BYTE peGreen; // 绿色分量（8 位）
  BYTE peBlue;  // 蓝色分量（8 位）
  BYTE peFlags; // 标记在窗口模式下，开始 10 种和最后 10 种颜色为 PC_EXPLICIT
          // 在全屏模式下，全部为 PC_NOCOLLAPSE
} PALETTEENTRY;
```

最后，如果使用的是 256 色模式，将加载 PALDATA2.PAL 中的调色板数据。我喜欢使用这个调色板，它有很好的颜色空间覆盖。您可以打开 PALDATA2.BMP 来查看这个调色板。当然，也可以从磁盘加载自己的 256 色调色板或加载某个位图附带的调色板。

4. 锁定/解锁功能映射

现在看一下用于锁定/解锁主表面和辅助表面的 4 个函数。在虚拟计算机接口中，它们类似于这样：

```
Lock_Primary(UCHAR **primary_buffer, int *primary_pitch);
Unlock_Primary(UCHAR *primary_buffer);

Lock_Secondary(UCHAR **secondary _buffer, int *secondary _pitch);
Unlock_Secondary(UCHAR *secondary _buffer);
```

在实际的图形库中，实现这种功能的函数如下：

```
UCHAR *DDraw_Lock_Primary_Surface(void);
int DDraw_Unlock_Primary_Surface(void);

UCHAR *DDraw_Lock_Back_Surface(void);
int DDraw_Unlock_Back_Surface(void);
```

实际函数惟一不同的地方是，它们直接修改下面的全局变量：

```
UCHAR *primary_buffer;   // 主视频缓存
UCHAR *back_buffer;      // 辅助缓存
int  primary_lpitch;     // 内存行跨距
int  back_lpitch;        // 内存行跨距
```

因此不需要任何参数。

5. 动画功能映射

对于映射，我想提到的最后一个函数是用于从辅助缓存复制到主缓存的动画函数。读者可能还记得，我们在虚拟计算机软件接口模型中，将它称为 Flip_Display()。该函数使得流畅的动画成为可能。在实际库中，执行这项任务的函数是：

```
int DDraw_Flip(void);
```

只是名称不同而已。该函数实现了虚拟模型所需要的功能。DDraw_Flip()函数对于全屏应用程序和窗口应用程序都适用。

有关虚拟计算机模型到真实函数的映射就介绍到这里。我只想让您明白我们的目标：创建一个通用图形系统，有一个 API 用于抽象图形渲染，让您只需要考虑两个帧缓存和几个函数。

3.7.2 最终的 T3DLIB 游戏控制台

下面列出完整的游戏控制台，我们将使用它来创建所有演示程序和游戏程序。这个升级后的游戏控制台名为 T3DCONSOLE2.CPP，其源代码如下。

```
// T3DCONSOLE2.CPP：本书使用的模板
// 编写应用程序时，可以将它用作模板
// 您可能想修改应用程序（如果是窗口模式）的分辨率，接管的 Directinput 设备等
// 该应用程序创建一个 640×480×16 窗口
// 因此运行该应用程序之前，必须将桌面设置为 16 位颜色模式
// 要使用全屏模式，只需在下面的#defines 中将 WINDOWED_APP 的值改为 FALSE(0)即可
// 同样，要修改位深，只需在函数 Game_Init()中相应地修改函数调用 DDraw_Init()即可
```

```
// 要编译该应用程序，务必在工程的链接列表中包括 DDRAW.LIB、DSOUND.LIB、
// DINPUT.LIB 和 WINMM.LIB
// 当然，还应将 C++源代码模块 T3DLIB1.CPP、T3DLIB2.CPP 和 T3DLIB3.CPP
// 以及头文件 T3DLIB1.H、T3DLIB2.H 和 T3DLIB3.H 存储在编译器的工作目录中

// 包含文件 ////////////////////////////////////////////////

#define INITGUID       // 确保所有的 COM 接口可用
            // 也可以不这样做，而是包含.LIB 文件 DXGUID.LIB

#define WIN32_LEAN_AND_MEAN

#include <windows.h>   // 包含重要的 Windows 头文件
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // 包含重要的 C/C++ 头文件
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h>  // 包含 DirectX 头文件
#include <dsound.h>
#include <dmksctrl.h>
#include <dmusici.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <dinput.h>
#include "T3DLIB1.h" // 包含游戏库头文件
#include "T3DLIB2.h"
#include "T3DLIB3.h"

// 常量 ////////////////////////////////////////////////////

// 用于 Windows 接口的常量
#define WINDOW_CLASS_NAME "WIN3DCLASS"  // 类名
#define WINDOW_TITLE    "T3D Graphics Console Ver 2.0"
#define WINDOW_WIDTH    640   // 窗口的大小
#define WINDOW_HEIGHT   480

#define WINDOW_BPP     16    // 窗口的位深(8、16、24 等)
                // 如果采用窗口模式，但不是全屏
                // 位深必须与系统位深相同
                // 另外，如果位深为 8 位
                // 将创建一个调色板，并将其与应用程序关联起来

#define WINDOWED_APP      1     // 0 表示非窗口模式，1 表示窗口模式

// 函数原型 ////////////////////////////////////////////////

// 游戏控制台
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// 全局变量 ////////////////////////////////////////////////

HWND main_window_handle      = NULL; // 用于存储窗口句柄
HINSTANCE main_instance      = NULL; // 用于存储实例
char buffer[256];                 // 用于打印文本
```

```
// 函数 ////////////////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
                            UINT msg,
                            WPARAM wparam,
                            LPARAM lparam)
{
// 这是系统的主消息处理程序
PAINTSTRUCT   ps;     // 用于WM_PAINT中
HDC     hdc;      // 设备场境句柄

// 判断是什么消息
switch(msg)
  {
  case WM_CREATE:
    {
  // 在这里添加初始化代码
  return(0);
  } break;

  case WM_PAINT:
    {
    // 开始绘制
    hdc = BeginPaint(hwnd,&ps);

    // 结束绘制
    EndPaint(hwnd,&ps);
    return(0);
   } break;

  case WM_DESTROY:
    {
    // 关闭应用程序
    PostQuitMessage(0);
    return(0);
    } break;

  default:break;

  } // end switch

// 处理其他消息
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc

// WINMAIN ////////////////////////////////////////////////

int WINAPI WinMain( HINSTANCE hinstance,
      HINSTANCE hprevinstance,
      LPSTR lpcmdline,
      int ncmdshow)
{
// 这是 winmain 函数

WNDCLASS winclass;  // 用于存储将创建的 Windows 类
HWND    hwnd;      // 窗口句柄
MSG    msg;      // 消息
HDC    hdc;    // 设备常境
PAINTSTRUCT ps;    // paintstruct

// 设置窗口类结构
winclass.style  = CS_DBLCLKS | CS_OWNDC | CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc   = WindowProc;
winclass.cbClsExtra   = 0;
```

```
winclass.cbWndExtra   = 0;
winclass.hInstance    = hinstance;
winclass.hIcon    = LoadIcon(NULL, IDI_APPLICATION);
winclass.hCursor  = LoadCursor(NULL, IDC_ARROW);
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL;
winclass.lpszClassName  = WINDOW_CLASS_NAME;

// 注册窗口类
if (!RegisterClass(&winclass))
  return(0);

// 创建窗口，根据 WINDOWED_APP 的值选择合适的窗口标记
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME,  // 窗口类
        WINDOW_TITLE,    // 标题
        (WINDOWED_APP ? (WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION) :
              (WS_POPUP | WS_VISIBLE)),
         0,0,    // 初始位置
         WINDOW_WIDTH,
              WINDOW_HEIGHT,  // 初始宽度和高度
         NULL,    // 父窗口句柄
        NULL,    // 菜单句柄
        hinstance,// 实例
        NULL)))   // 额外的创建参数
return(0);

// 将窗口句柄和实例存储到全局变量中
main_window_handle = hwnd;
main_instance    = hinstance;

// 调整窗口大小，使客户区域的大小为 width × height
if (WINDOWED_APP)
{
// 调整窗口的大小，使客户区域的大小等于请求的大小
// 如果应用程序为窗口模式时，将由边框和控件调整
// 应用程序不是窗口模式时，无需调整
RECT window_rect = {0,0,WINDOW_WIDTH-1,WINDOW_HEIGHT-1};

// 调用函数来调整 window_rect
AdjustWindowRectEx(&window_rect,
   GetWindowStyle(main_window_handle),
   GetMenu(main_window_handle) != NULL,
   GetWindowExStyle(main_window_handle));

// 将客户区域偏移量保存到全局变量中，因为 DDraw_Flip()需要使用它们
window_client_x0 = -window_rect.left;
window_client_y0 = -window_rect.top;

// 调用 MoveWindow()来移动窗口
MoveWindow(main_window_handle,
      0, // 水平位置
      0, // 垂直位置
      window_rect.right - window_rect.left, // 宽度
      window_rect.bottom - window_rect.top, // 高度
      FALSE);

// 显示窗口
ShowWindow(main_window_handle, SW_SHOW);
} // end if windowed

// 执行游戏控制台特有的初始化
Game_Init();

// disable CTRL-ALT_DEL,ALT_TAB ,comment this line out
// if it causes your system to crash
SystemParametersInfo(SPI_SCREENSAVERRUNNING, TRUE, NULL, 0);
```

```
// 进入主事件循环
while(1)
   {
   if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
  // 检查是否是退出消息
    if (msg.message == WM_QUIT)
      break;

  // 转换加速键
  TranslateMessage(&msg);

  // 将消息发送给 window proc
  DispatchMessage(&msg);
  } // end if

    // 主游戏处理函数
    Game_Main();

  } // end while

// 关闭游戏并释放所有的资源
Game_Shutdown();

//禁用 CTRL-ALT_DEL 和 ALT_TAB
// 如果该行代码导致系统崩溃,将其注释掉
SystemParameters Info(SPI_SCREENSAVER RUNNING,TRUE,NULL,0);

// 返回到 Windows 操作系统
return(msg.wParam);

} // end WinMain

// T3DII 游戏编程控制台函数 ////////////////

int Game_Init(void *parms)
{
// 所有的游戏初始化工作都在这个函数中执行

// 启动 DirectDraw
DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP, WINDOWED_APP);

// 初始化 directinput
DInput_Init();

// 接管键盘
DInput_Init_Keyboard();

// 在这里加入接管其他 directinput 设备的函数调用...


// 初始化 directsound 和 directmusic
DSound_Init();
DMusic_Init();

// 隐藏鼠标
ShowCursor(FALSE);

// 随机数生成器
srand(Start_Clock());

// 在这里加入初始化代码...


// 成功返回
return(1);
```

```
} // end Game_Init

//////////////////////////////////////////////////////////////////////

int Game_Shutdown(void *parms)
{
// 在这个函数中关闭游戏并释放为游戏分配的所有资源

// 关闭一切

// 在这里加入释放为游戏分配的资源的代码....


// 关闭 directsound
DSound_Stop_All_Sounds();
DSound_Delete_All_Sounds();
DSound_Shutdown();

// 关闭 directmusic
DMusic_Delete_All_MIDI();
DMusic_Shutdown();

// 释放所有输出设备
DInput_Release_Keyboard();

//关闭 directinput
DInput_Shutdown();

// 最后关闭 directdraw
DDraw_Shutdown();

// 成功返回
return(1);
} // end Game_Shutdown

//////////////////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
// 这是游戏的核心，将不断地被实时调用
// 它类似于 C 语言中的 main( )，所有游戏调用都是在这里进行的

int index; // 循环变量

// 启动定时时钟
Start_Clock();

// 清空缓存
DDraw_Fill_Surface(lpddsback, 0);

// 读去键盘和其他设备输入
DInput_Read_Keyboard();

// 在这里加入游戏逻辑代码...



// 交换缓存
DDraw_Flip();

// 同步到 30 帧/秒
Wait_Clock(30);

// 检查用户是否要退出
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);
```

```
  } // end if

// 成功返回
return(1);

} // end Game_Main
```

下面来看一看游戏控制台的关键部分，然后开始使用它。首先，您可能注意到了，有些代码使用粗体显示，以指出它们很重要。下面来分析一下这些代码的功能。

1. 打开游戏控制台窗口

第一个以粗体显示的代码段位于#defines 部分：

```
#define WINDOW_WIDTH   640  // 窗口大小
#define WINDOW_HEIGHT   480

#define WINDOW_BPP    16  //  窗口的位深(8、16、24 等)
                  // 如果采用窗口模式，但不是全屏
                  // 位深必须与系统位深相同
                  // 另外，如果位深为 8 位
                  // 将创建一个调色板，并将其与应用程序关联起来

#define WINDOWED_APP   1   // 0 表示非窗口模式，1 表示窗口模式
```

这些代码非常重要，因为它们控制窗口的大小（或全屏大小）、位深以及采用窗口显示还是全屏显示。当前，窗口大小被设置为 640×480、每个像素 16 位、使用窗口显示。代码的很多地方都使用了这些参数，其中两个最重要的地方是，在 WinMain()函数中创建窗口时和在 Game_Main()函数中调用 DDraw_Init()时。先来看一下 WinMain()中的调用。

如果您看一下 WinMain()中的 Create_Window()函数调用（如下所示），您将发现其中有一个三元条件运算符，它用于检测应用程序是否使用窗口模式。

```
// 创建窗口，根据 WINDOWED_APP 的值选择合适的窗口标记
if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME, // class
        WINDOW_TITLE,   // title
        (WINDOWED_APP ? (WS_OVERLAPPED | WS_SYSMENU | WS_CAPTION) :
              (WS_POPUP | WS_VISIBLE)),
         0,0,    // 位置
         WINDOW_WIDTH, // 宽度
              WINDOW_HEIGHT, // 高度
         NULL,    // 父窗口句柄
        NULL,    // 菜单句柄
        hinstance,// 实例
        NULL)))  // 创建参数
return(0);
```

然后使用适当的 Windows 标记来创建窗口。如果要使用窗口显示，且希望看到窗口边框和控制（如“关闭”框），应使用 Windows 标记（WM_OVERLAPPED|WM_SYSMENU|WS_CAPTION）。

另一方面，如果在代码中选择使用全屏模式，将使用与 DirectDraw 表面相同的大小来创建窗口，且窗口中没有控件；因此使用窗口样式 WM_POPUP。另外，窗口被创建后，下一段代码极其棘手。实质上，它调整窗口尺寸，使窗口客户区域与请求的尺寸相同，而不是请求的尺寸减去边框和控件的尺寸。

在 Windows 编程中，当您创建一个尺寸为 WINDOW_WIDTH × WINDOW_HEIGHT 的窗口时，并不意味着窗口的客户区域尺寸为 WINDOW_WIDTH × WINDOW_HEIGHT。它意味着整个窗口的尺寸为 WINDOW_WIDTH × WINDOW_HEIGHT。因此，如果没有边框和控件，客户区域的尺寸将为 WINDOW_WIDTH × WINDOW_HEIGHT，但如果窗口有控件，客户区域将会小一些，如图 3.13 所示。

为解决这种问题，需要动态地调整窗口的大小，使窗口的客户区域尺寸正好为请求的窗口尺寸。下面是完成该项任务的代码：



图 3.13　整个窗口与客户区域之比较

```
// 调整窗口大小，使客户区域的大小为 width × height
if (WINDOWED_APP)
{
// 调整窗口的大小，使客户区域的大小等于请求的大小
// 这是由于应用程序为窗口模式时，将由边框和控件调整
// 应用程序不是窗口模式时，无需调整
RECT window_rect = {0,0,WINDOW_WIDTH-1,WINDOW_HEIGHT-1};

// 调用函数来调整 window_rect
AdjustWindowRectEx(&window_rect,
   GetWindowStyle(main_window_handle),
   GetMenu(main_window_handle) != NULL,
   GetWindowExStyle(main_window_handle));

// 将客户区域偏移量保存到全局变量中，因为 DDraw_Flip()需要使用它们
window_client_x0 = -window_rect.left;
window_client_y0 = -window_rect.top;

// 调用 MoveWindow()来移动窗口
MoveWindow(main_window_handle,
      0, // 水平位置
      0, // 垂直位置
      window_rect.right - window_rect.left, // 宽度
      window_rect.bottom - window_rect.top, // 高度
      FALSE);

// 显示窗口
ShowWindow(main_window_handle, SW_SHOW);
} // end if windowed
```

还有其他两种方法可用于完成这种任务目标，但是我通常使用前面介绍的方法。另一种方法是，了解窗口有哪些控件，然后向 Windows 查询这种控件的尺寸，再计算窗口应有的总尺寸。不管采用哪种方法，都应确保选择窗口模式时，窗口的客户区域为 WINDOW_WIDTH × WINDOW_HEIGHT。

注意：如果将 WINDOWED_APP 设置为 0，将不会执行上述逻辑，因为 WM_POPUP 窗口不包含任何控件，因此无需调整窗口的大小。

创建窗口并根据需要调整其大小后，将调用函数 Game_Init()。创建自己的应用程序时，您将在该函数中加入自己的初始化代码；但当前它完成的工作很少，只是调用函数 DDraw_Init()。

```
// 启动 DirectDraw
DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP, WINDOWED_APP);
```

注意：您可能已经注意到，在主事件循环附近的 SystemParametersInfo( )函数调用。这些调用使 Windows 认为屏幕保护程序正在运行，从而忽略按键 Alt+Tab。如果您没有正确地处理 Alt+Tab，DirectX 应用程序可能会崩溃，调用该函数后您不需要再考虑这些问题。如果有兴趣，可以查看 DirectX SDK。一般而言，当应用程序失去焦点时，将需要恢复所有丢失的表面，并重新接管所有输入设备，这是令人痛苦的！

2. 使用和编译游戏控制台

T3DCONSOLE2.CPP 只是一个“外壳”或模板，您将使用它来创建应用程序。您可以使用它来处理所有与 Windows 相关的问题，这样，只需在游戏函数 Game_Init()、Game_Main()和 Game_Shutdown()中添加所需的代码即可。然而，要编译这个游戏控制台，需要包括下列文件：

- T3DLIB1.CPP|H——DirectDraw 模块；
- T3DLIB2.CPP|H——DirectInput 模块；
- T3DLIB3.CPP|H——DirectSound 和 DirectMusic 模块。

下列文件必须位于您的根目录中：

- PALDATA1|2.PAL——256 色模式的默认调色板。

此外，还必须链接下列库文件：

- DDRAW.LIB；
- DSOUND.LIB；
- DINPUT.LIB；
- DINPUT8.LIB。

请将编译器设置为生成 Win32 .EXE 目标程序。别忘了在链接列表中添加 Direct .LIB 文件，并设置编译器的搜索路径，使它能够找到所有头文件。

我在没有添加任何内容的情况下，编译了该游戏控制台，创建文件 T3DCONSOLE2.EXE。该应用程序创建一个 640×480×16 的窗口。要运行该程序，必须将桌面设置为 16 位颜色模式。

警告：基于游戏控制台的窗口应用程序不会修改桌面的位深。如果您愿意，也可以修改桌面的位深，但可能产生灾难性后果。原因是：如果运行了其他应用程序，然后启动一个 DirectDraw 应用程序，它修改屏幕的位深，但创建一个非全屏的窗口，其他正在运行的应用程序将不能正常工作，并可能导致系统崩溃。对于全屏应用程序，您可以使用自己想要的任何分辨率和位深；但对于窗口应用程序，应该测试当前位深，看它是否满足您的需求，然后再启动您的应用程序，并让用户能够切换桌面。

只需修改一条#define 语句，就可以将应用程序改成全屏的 640×480×16 应用程序：

```
#define WINDOWED_APP   0   // 0表示非窗口模式，1表示窗口模式
```

然而，窗口应用程序更有趣，也更容易调试，因此本书主要使用窗口应用程序。尽管如此，游戏程序能够正常运行后，建议您还是将其更改为全屏模式。

3.8 范例 T3LIB 应用程序

下面介绍几个范例应用程序，它们是在游戏控制台 T3DCONSOLE2.CPP 的基础上编写而成的，演示了如何处理图形、声音、音乐和输入。这些演示程序都有些实际性内容，它们都来自《Windows 游戏编程大师技巧》，但进行了更新。这样，您看到的将是使用控制台模板的真实范例。然而，对声音和音乐的演示则没有原来那么优美。

最后，本书将逐步完善 T3DLIB 引擎，添加模块 T3DLIB4、5、6 等。

3.8.1 窗口应用程序

T3DCONSOLE2.CPP 本身就是一个真正的 16 位窗口应用程序，但为创建演示程序，我将在其中添加一些内容。

我将《Windows 游戏编程大师技巧》中的人工智能演示程序（模式系统演示程序）转换为一个 16 位窗口程序。在附带光盘中，该程序名为 DEMOII3_1.CPP|EXE，图 3.14 是该程序运行时的屏幕截图。要运行该程序，桌面必须是 16 位颜色模式下。另外，要编译它，必须包括所有的库模块。

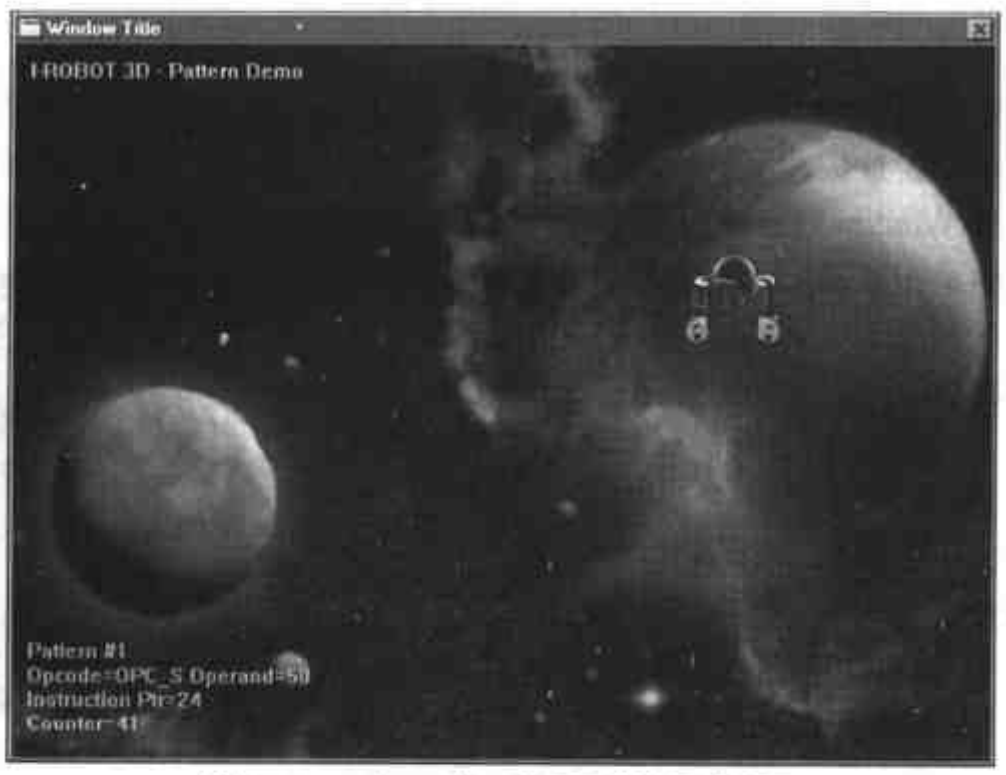


图 3.14　一个 AI 窗口演示程序的屏幕截图

该程序不仅演示了 16 位窗口模式，还演示了如何加载位图、使用 BOB 库、播放声音以及 16 位图形模式下的其他操作。通过研究其源代码，读者将可以学习很多知识。

3.8.2　全屏应用程序

使用我们的库模块编写全屏应用程序没有什么不同：实际上更容易，因为发生资源共享问题的可能性更小。在 256 色模式下，您可以完全控制整个调色板，而不像在 256 色窗口模式下那样，不能修改调色板的前 10 种和后 10 种颜色。另外，在全屏模式下，可以切换到任何位深和分辨率，而不用考虑桌面的位深。

为编写全屏应用程序，我将《Windows 游戏编程大师技巧》中一个 256 色物理学演示程序的代码集成到我们的游戏控制台模板中。该程序名为 DEMOII3_2.CPP|EXE，图 3.15 是它运行时的屏幕截图。该程序再次演示了如何使用位图、BOB 和 256 色调色板模式。

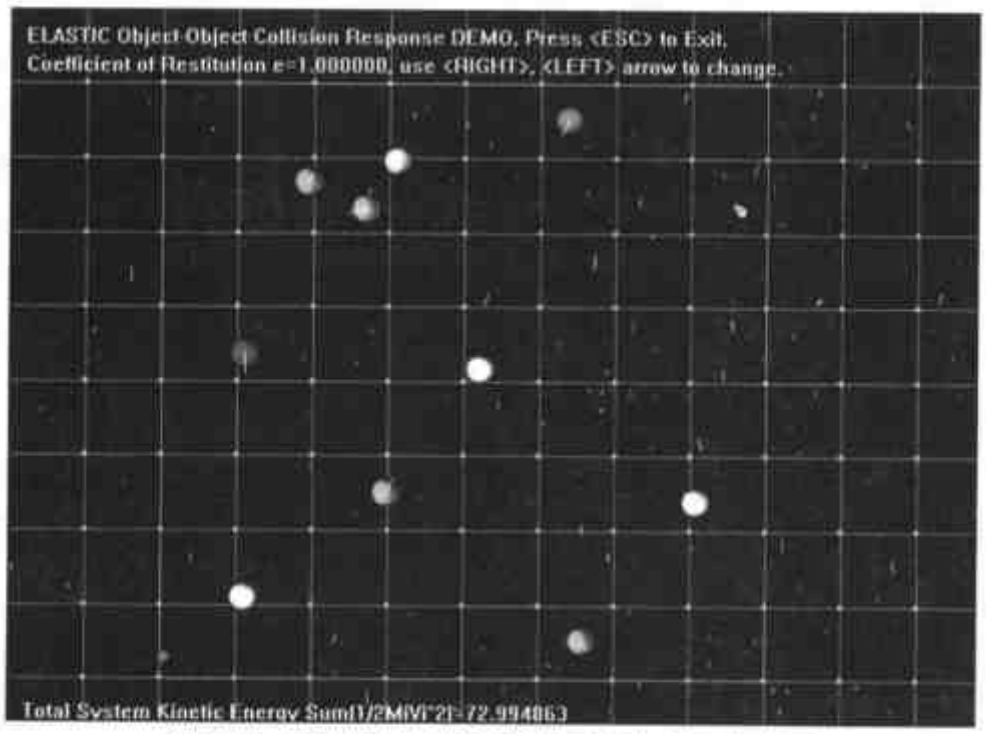


图 3.15　256 色全屏物理学演示程序的屏幕截图

3.8.3 声音和音乐

在 DirectSound 和 DirectMusic 下，使用声音和音乐并不难，但也并不容易。这些系统非常复杂，有很多功能。然而，我们只是想从磁盘加载声音和音乐，然后播放它们！T3DLIB2.CPP|H 模块以非常简洁的方式处理这些任务。

声音和音乐演示程序在图形方面非常简单，这样您可以将重点放在使声音和音乐能够正常运行的函数调用上。我采用《Windows 游戏编程大师技巧》中一个同时使用 DirectSound 和 DirectMusic 的演示程序，它是一个包含菜单的简单 Windows 程序，用户通过菜单选择要播放的 MIDI 歌曲，同时可以触发各种声音效果。另外，该演示程序是在精简的游戏控制台的基础上编写的，旨在最大限度地删除无关的东西。



图 3.16 声音和音乐演示程序的屏幕截图

该演示程序名为 DEMOII3_3.CPP|EXE，图 3.16 是它运行时的屏幕截图。这个程序使用了两三种资源，如菜单和光标，您需要将相应的文件加入到工程中：

- DEMOII3_3.RC——Windows 菜单资源文件；
- DEMOII3_3_RES.H——包含资源标识符的头文件；
- T3DX.ICO——该程序使用的光标图标。

同样，它也是一个 DirectX 应用程序，因此您需要在工程中包括 DirectX .LIB 文件。从技术上说，该程序并不需要 DDRAW.LIB，但为安全起见，应该总是在链接文件列表中包括所有的 DirectX 库文件。该程序没有调用 T3DLIB1.CPP 和 T3DLIB2.CPP 中的任何图形或输入函数，因此在工程中不要包括这两个.CPP 文件，否则将出现链接错误！

3.8.4 处理输入

最后一个范例是输入演示程序。您知道，游戏使用的主要输入设备有 3 种：

- 键盘；
- 鼠标；
- 游戏杆。

当前，游戏杆可以指很多设备，但通常将除键盘和鼠标之外的所有输入设备都称为游戏杆。

使用输入库相当简单，它包含在文件 T3DLIB2.CPP|H 中，支持键盘、鼠标和游戏杆设备。只要几个函数调用，就可以初始化 DirectInput、接管输入设备，然后在主事件循环中从输入设备中读取数据。这里将为每种输入设备提供一个演示程序：键盘、鼠标和游戏杆。当然，也可以同时使用所有的输入设备，但演示程序将分开进行，让您能够更好地理解如何使用输入库。

1. 键盘演示程序

同样，该演示程序也将以《Windows 游戏编程大师技巧》中的代码为基础。键盘演示程序名为 DEMOII3_4.CPP|EXE，图 3.17 是它运行时的屏幕截图。该程序让用户能够使用键盘在屏幕上移动 BOB。如果您查看读取键盘输入的代码，将发现返回的数据结构是一个 256 字节的布尔型数组，每个元素代表一个键。这个数组如下所示：

```
UCHAR keyboard_state[256]; // 键盘状态数组
```

要访问这个数组，需要使用 DirectInput 键盘常量，它们的格式为 DK_*。前面的表 3.2 列出了一些最常用的键盘常量。您只需检测哪个键被按下，调用函数 DInput_Read_Keyboard()取回键盘数据后，就可以使用如下测试：

```
if (keyboard_state[DIK_UP])
  {
  // 上箭头键被按下是执行的代码
  } // end if
```

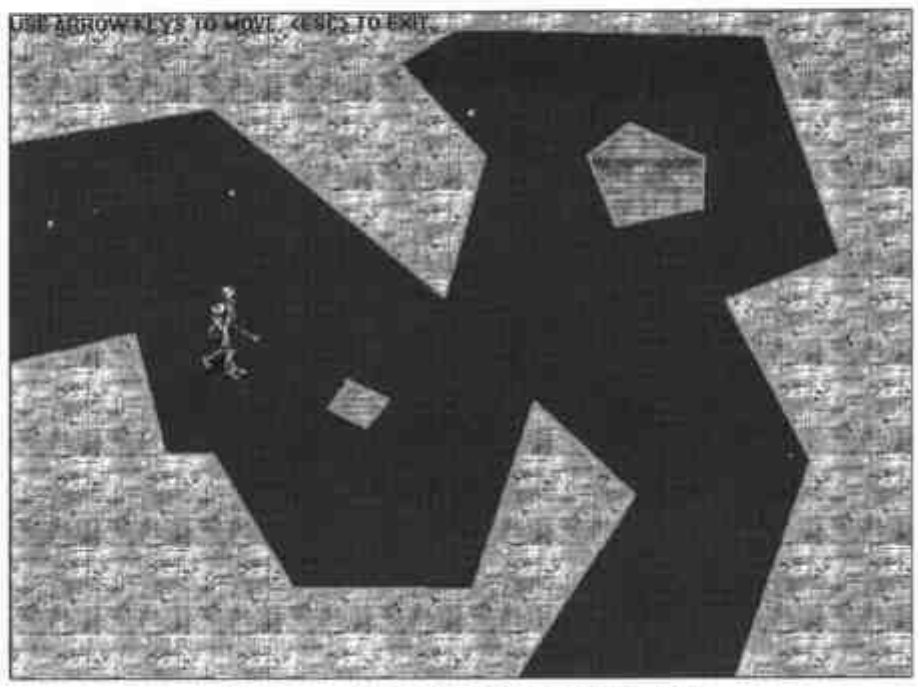


图 3.17　键盘演示程序的屏幕截图

要编译这个演示程序，需要所有的 DirectX .LIB 文件、T3DLIB1.CPP|H、T3DLIB2.CPP|H 和 T3DLIB3.CPP|H。

2. 鼠标演示程序

鼠标演示程序也是以《Windows 游戏编程大师技巧》中的代码为基础的。它是一个很小的绘图程序，如图 3.18 所示。该程序名为 DEMOII3_5.CPP|EXE，它演示有关 GUI 编程的很多知识，如对象拾取、鼠标跟踪等。除鼠标跟踪代码外，绘图代码也值得学习。

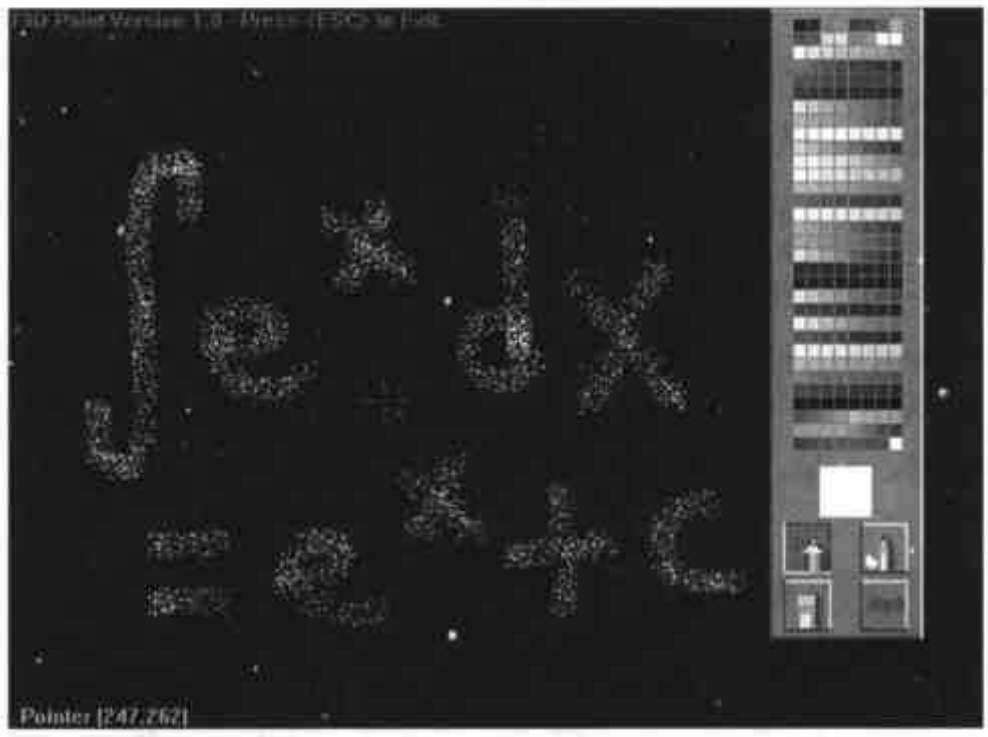


图 3.18　基于鼠标的绘图程序的屏幕截图

鼠标输入设备可以以两种模式运行：绝对模式和相对模式。我更喜欢使用相对模式，因为总是可以计算出鼠标在什么位置，因此 API 封装函数将鼠标系统设置为相对模式。每次读取鼠标数据时，都将获得相对于上一次的偏移量。每次调用 DInput_Read_Mouse() 函数时，都将读取鼠标数据，并将其存储到下面的全局鼠标记录中：

```
DIMOUSESTATE mouse_state; // 鼠标状态
该 DIMOUSESTATE 对象的结构如下：
typedef struct DIMOUSESTATE {
  LONG lX; // x 坐标
  LONG lY; // y 坐标
  LONG lZ; // z 坐标
  BYTE rgbButtons[4]; // 鼠标按钮的状态
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

要编译这个演示程序，需要所有的 DirectX .LIB 文件、T3DLIB1.CPP|H、T3DLIB2.CPP|H 和 T3DLIB3.CPP|H。

3. 游戏杆演示程序

在当今的计算机上，“游戏杆”设备各种各样——从游戏键盘到方向盘（paddle），但是任何游戏杆类的设备无非是两种输入设备的集合：

- 模拟轴；
- 按钮。

图 3.19 显示了 4 种不同的游戏杆类设备：典型的游戏杆、高级游戏杆、游戏键盘（game pad）和方向盘。您可能不认为它们都是游戏杆，但确实都是。游戏杆有两个轴：x 轴和 y 轴，还有两个按钮；游戏键盘全部由按钮组成；而方向盘有一个旋转轴和两三个按钮。



图 3.19 4 种基本的游戏杆设备

当然，在 DirectInput 中，有很多常量用于定义一些最流行的游戏杆类型，如普通游戏杆、飞行杆（flight stick）、方向盘、游戏键盘等，但它们的使用方式通常相同：从模拟滑块获取连续变化的数据或大量表示按钮 on/off 的数字信号。DInput_Read_Joystick() 函数返回的数据存储在下面的全局变量中：

```
DIJOYSTATE joy_state;   // 游戏杆状态
```

其结构如下：

```
typedef struct DIJOYSTATE {
  LONG  lX; // x 坐标
  LONG  lY; // y 坐标
  LONG  lZ; // z 坐标
  LONG  lRx; // 绕 x 轴旋转角度
  LONG  lRy; // 绕 y 轴旋转角度
  LONG  lRz; // 绕 z 轴旋转角度
  LONG  rglSlider[2]; // u、v 滑块（slider）的位置
  DWORD  rgdwPOV[4];  // 视点状态
  BYTE  rgbButtons[32]; // 按钮 0～31 的状态
} DIJOYSTATE, *LPDIJOYSTATE;
```

用于演示游戏杆设备的程序也取自《Windows 游戏编程大师技巧》，它是一个类似于 Centipede 的小型游戏（虽然不太像游戏）。在附带光盘中，该程序名为 DEMOII3_6.CPP|EXE，其屏幕截图如图 3.20 所示。同样，要编译这个演示程序，需要所有的 DirectX .LIB 文件以及 T3DLIB1.CPP|H、T3DLIB2.CPP|H 和 T3DLIB3.CPP|H。

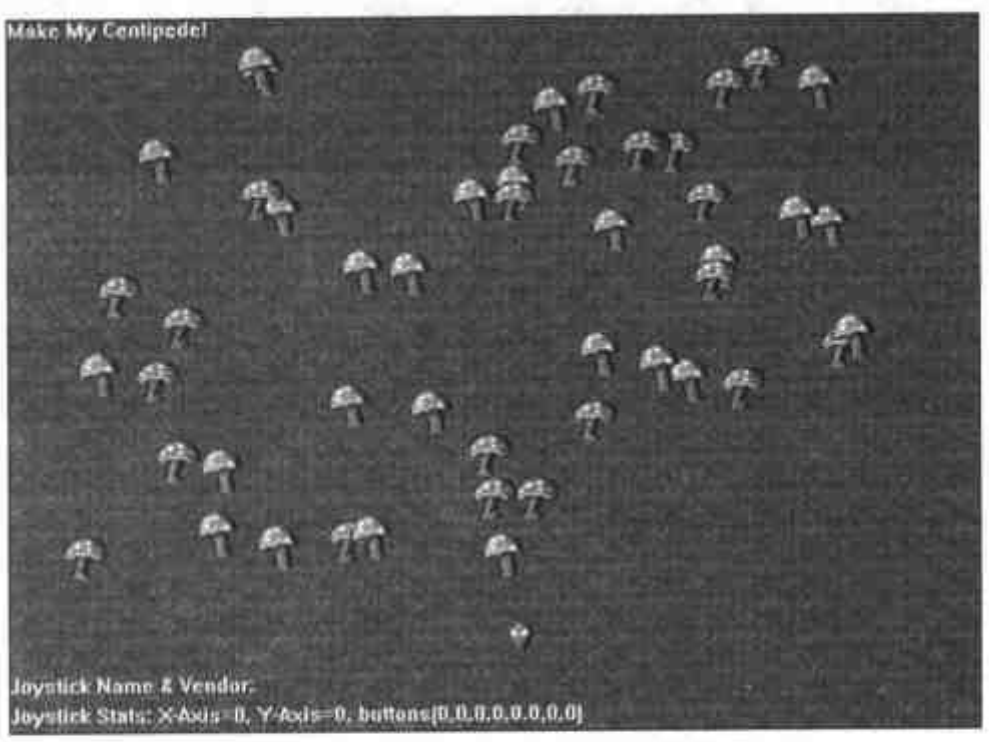


图 3.20 游戏杆演示程序的屏幕截图

注意：学习这些演示程序时，请注意与游戏编程相关的其他内容，如基本游戏逻辑、碰撞检测和武器系统等。

3.9 总　　结

本章在《Windows 游戏编程大师技巧》和本书之间搭建了一座桥梁。您应该知道，3D 图形学研究的不是 DirectX 和 Windows，而是数学、数学还是数学。为进行 3D 图形学编程，只需要一个包含一些函数和两个帧缓存的平台。即使您没有阅读过《Windows 游戏编程大师技巧》，也很容易地理解本书将使用的 API。您应充满信心，因为我们只使用几个函数来实现虚拟图形计算机，供本书其他章节使用。对于那些已经阅读过《Windows 游戏编程大师技巧》的读者，我相信您们会非常高兴 T3DLIB 现在支持 16 位窗口模式了！我也一样。

第二部分

3D数学和变换

第4章：三角学、向量、矩阵和四元数

第5章：建立数学引擎

第6章：3D图形学简介

第7章：渲染3D线框世界

第 4 章　三角学、向量、矩阵和四元数

以前编写计算机游戏时，只需要很少的三角学和代数方面的知识；但现在完全不同了。还记得几年前，我联系 *Game Developer Magazine* 的编辑，想发表一篇关于向量的文章，得到的答复是数学太高深了；而现在阅读这份杂志时，发现其中充斥着高深的向量和微积分方面的内容！本章复习学习 3D 计算机图形学时所需的基本数学知识。不可能在一章中介绍大学四年的全部数学知识，而只能蜻蜓点水地复习一下，如果您要了解更详细的内容，可参考相应的数学书籍。当然，如果您已经是数学专家，可跳过本章，直接进入创建数学库的下一章。本章介绍下列内容：

- 数学表示法；
- 坐标系；
- 三角学；
- 向量；
- 矩阵和线性代数；
- 基本几何实体；
- 使用参数化方程；
- 四元数简介；
- 微积分基础。

4.1　数学表示法

数学像语言一样，由一组表示数学概念、公式、方程和运算的符号组成。高等数学存在的问题是，很多符号都是希腊字母。开始介绍数学知识之前，先列出一些常用的符号，说明探讨纯数学问题时本书采用的约定。虽然有些概念现在没有什么意义，但至少您可以提前了解一些定义。表 4.1 列出了我们将使用的基本数学类型。

表 4.1　　数学类型

类型	表示法	示例	类型	表示法	示例
标量	小写字母	a、b、x、y	向量的分量	<分量>	<ux，uy，uz>
无穷大	∞	$+\infty$，$-\infty$	行向量	[分量]	[ux，uy，uz]
角度	小写希腊字母	θ，ϕ，α，β	点	粗体小写字母	$\mathbf{a}$，$\mathbf{p}_1$，$\mathbf{p}_2$
向量	粗体小写字母	$\mathbf{u}$，$\mathbf{v}$，$\mathbf{a}_x$，$\mathbf{a}_y$	矩阵	粗体小写字母	$\mathbf{A}$，$\mathbf{B}$，$\mathbf{M}_2$
四元数	粗体小写字母	$\mathbf{p}$，$\mathbf{q}$，$\mathbf{r}_x$，$\mathbf{s}_y$	三角形	Δ加顶点	$\Delta\mathbf{P}_0\mathbf{P}_1\mathbf{P}_2$，$\Delta\mathbf{abc}$

注：所有类型都可以带下标或上标。

简要地介绍基本类型后，接下来介绍一些标准数学运算符，如表 4.2 所示。

表 4.2　　数学运算符

运算	表示法	示例	运算	表示法	示例
标量乘法	×、*	3×5，a*b	长度	|expr|	|u|
矩阵乘法	×	A×B	行列式	det（expr）	det(M)
点积	·	u · v	转置	上标 T	v^T
总和	Σexpr		积分	∫ expr	
绝对值	|expr|	|-35|	微分	'	x'，x''

注：“expr”表示表达式。

当然，后面可能介绍更多的表示法，但表 4.1 和表 4.2 是大多数数学书采用的表示规则。

警告：通常，大多数数学书用|M|来表示矩阵 M 的行列式，用||v||表示向量 v 的长度或范数。但这种双竖线符号不太好排版，所以本书使用 det（M）来表示矩阵的行列式，使用|v|表示向量的长度或范数。

4.2　2D 坐标系

规定一些数学表示法后，还必须确保作者和读者对几何学的认识一致。下面首先介绍常规 2D 坐标系，然后介绍更复杂的 3D 坐标系。

4.2.1　2D 笛卡尔坐标

笛卡尔坐标（Cartesian Coordinates）系是最常见的 2D 坐标系，每个人都使用过。笛卡尔坐标系基于两条相互垂直的坐标轴：x 轴和 y 轴，如图 4.1 所示。正 x 轴向右，负 x 轴向左；正 y 轴向上，负 y 轴向下。x 和 y 都为 0 点叫作原点。用工程术语说，y 轴也叫纵轴（ordinate），x 轴也叫横轴（abscissa）。

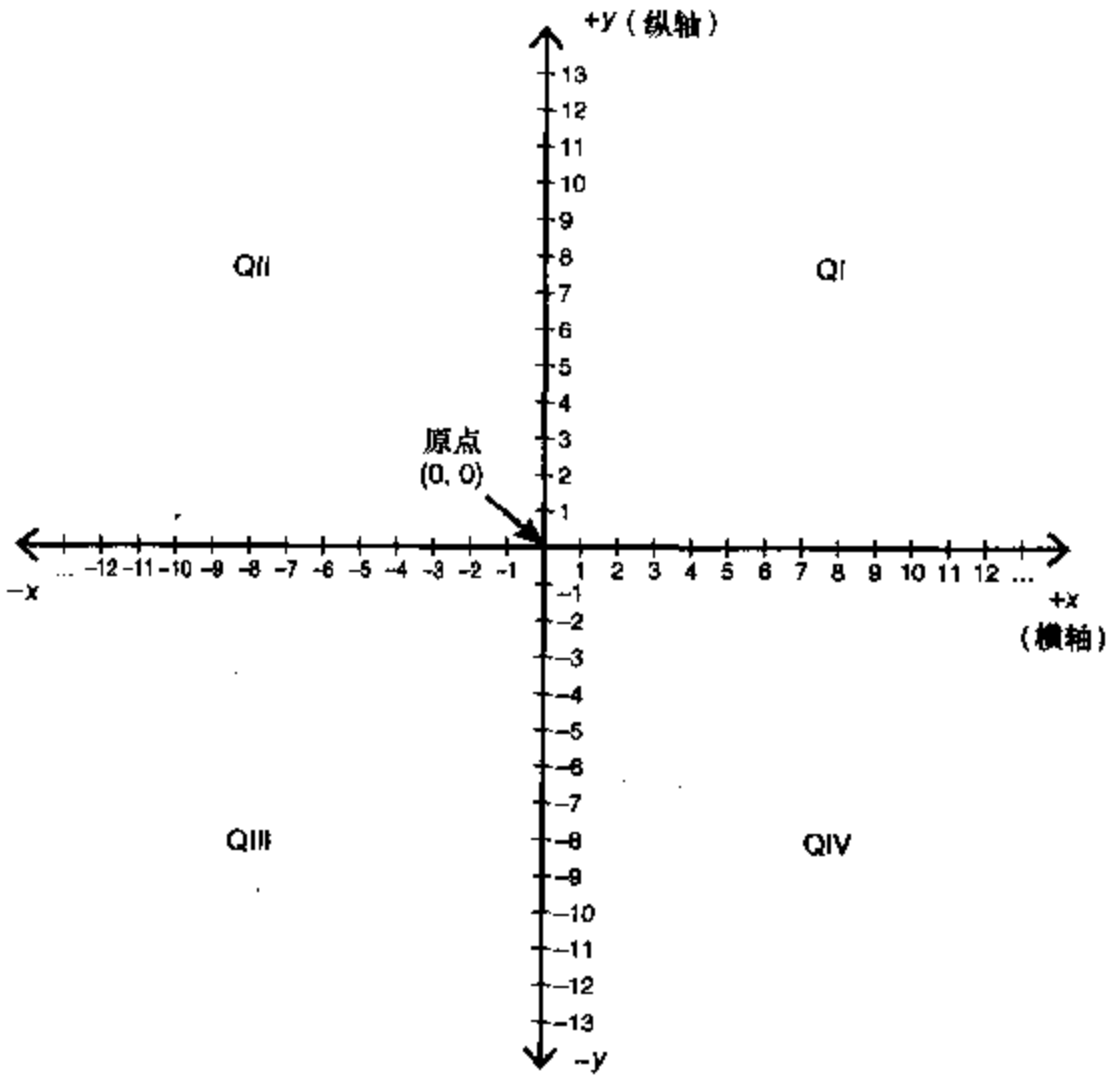


图 4.1　笛卡尔坐标系

2D *x*-*y* 坐标系中有 4 个象限，分别被标记为 QI、QII、QIII 和 QIV。这些象限通过 *x*、*y* 坐标的符号相区分。表 4.3 说明了每个象限中 *x*、*y* 坐标的符号。

表 4.3　　每个象限中的坐标符号

象限	*x* 坐标的符号	*y* 坐标的符号	象限	*x* 坐标的符号	*y* 坐标的符号
I	+	+	III	-	-
II	-	+	IV	+	-

提示：很多图形算法都以这样的方式进行优化：首先在 QI 中解决问题，然后根据对称性将解决方案反射到其他象限（在 3D 坐标系中为卦限）。

最后，要在 2D 笛卡尔坐标系中定义一个点，需要指定其 *x* 坐标和 *y* 坐标。例如，点 p(5，3) 意味着 *x* = 5，*y* = 3，如图 4.2 所示。

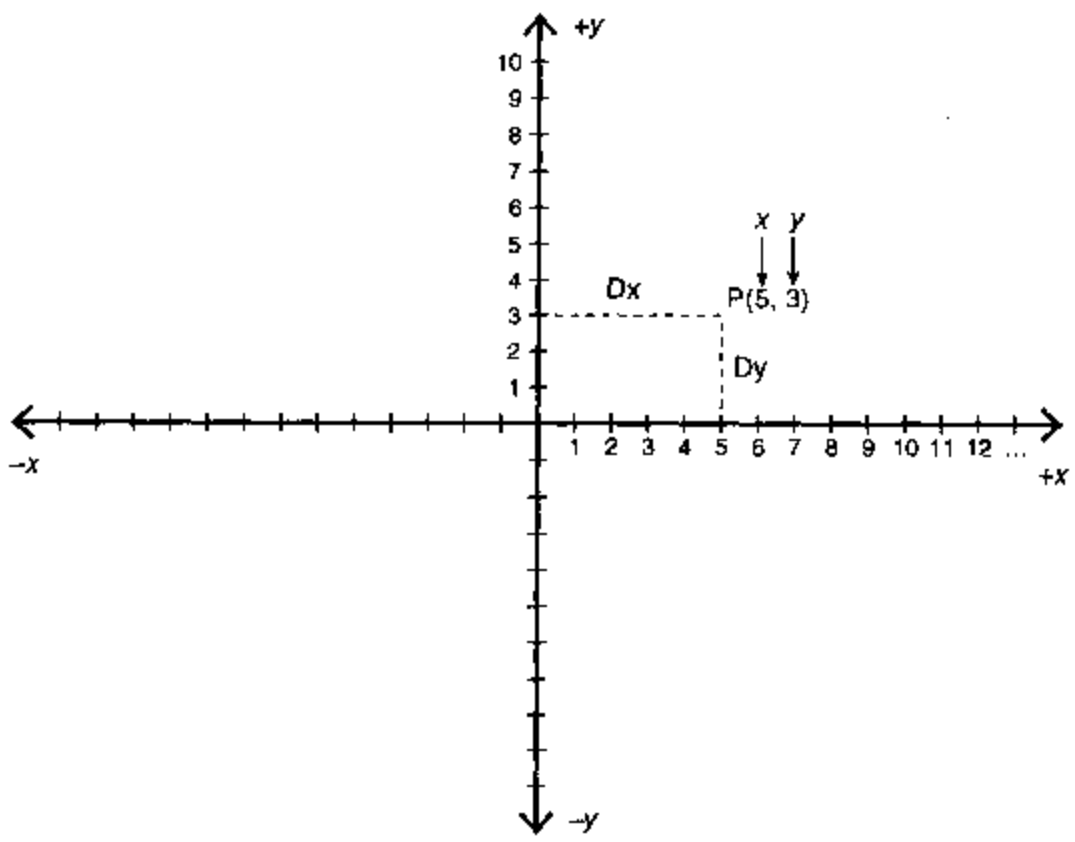


图 4.2　在笛卡尔坐标系中定义点

4.2.2　2D 极坐标

另一种支持两个自由度的坐标系是极坐标（polar coordinates）系。极坐标是游戏 Wolfenstein 和光线投射技术的基础。极坐标使用方向（heading）和距离来定义 2D 空间中的点，而不是使用 (x，y) 坐标。图 4.3 描述了标准的 2D 极坐标系，从中可知，使用两个变量来定义一个点：到原点（极点）的距离 *r*，方向（角度）θ。因此，**p**(*r*，θ) 表示点 **p** 位于这样的射线上：相对于参考线（通常为 *x* 轴）的反时针角度为 θ，且离原点的距离为 *r*。图 4.4 显示了点 $\mathbf{p}_1$(10，30) 和 $\mathbf{p}_2$(6，150)，并以 *x*-*y* 坐标系为参考。

在 2D 极坐标和笛卡尔坐标之间进行转换

虽然极坐标很好且很好看，但是有时候将极坐标转换为笛卡尔坐标也很有用。那么如何进行转换呢？在笛卡尔坐标和极坐标之间进行转换很容易，只需使用少量有关几何学和直角三角形的知识。图 4.5 显示了一个位于 2D 笛卡尔坐标系象限 I 中的标准直角三角形。对于直角三角形斜边的端点 P，可以使用下面的公式进行极坐标到笛卡尔坐标的转换。

```
x = r*cos(θ)
y = r*sin(θ)
```

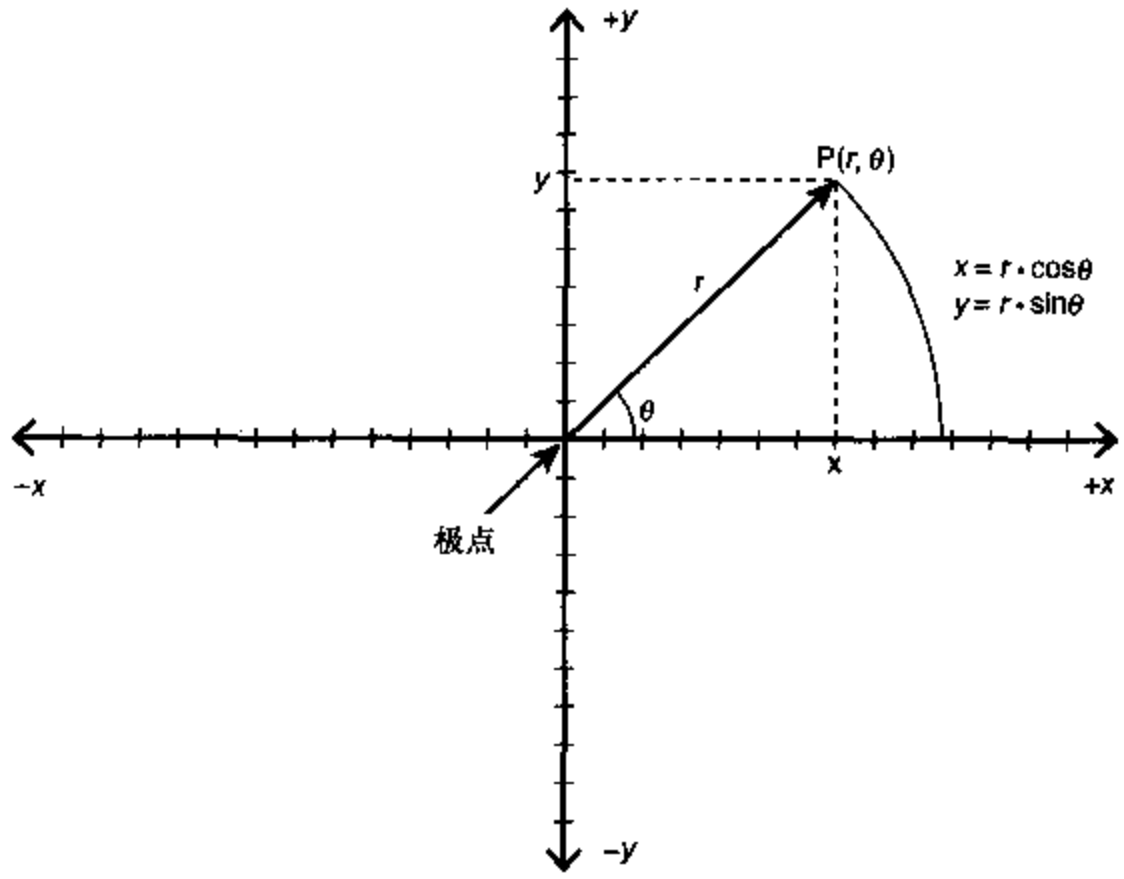


图 4.3　2D 极坐标系

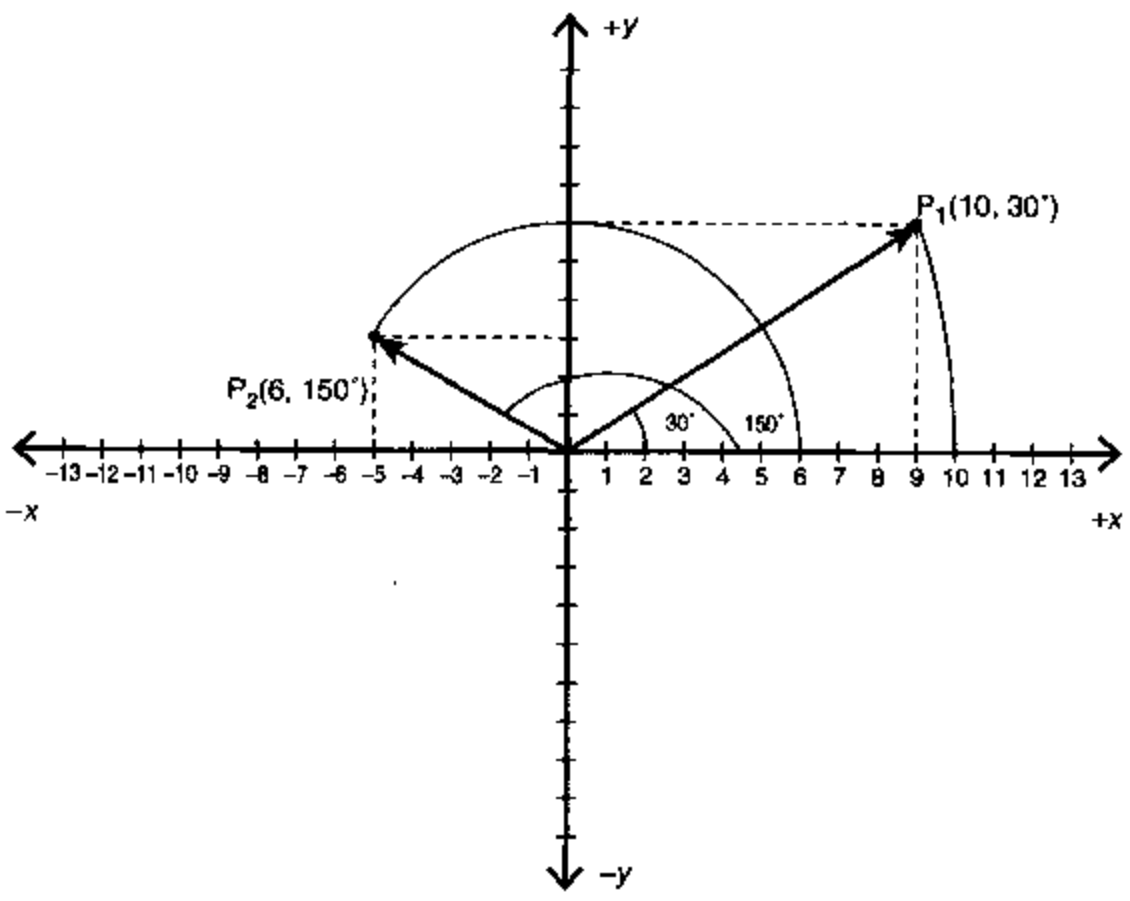


图 4.4　在 2D 极坐标系中定义点

提示：如果您不熟悉三角学，本章稍后将介绍它，现在只需使用该方程即可。

将笛卡尔坐标转换为极坐标稍微复杂些。问题在于，需要计算点 $\mathbf{p}(x, y)$ 与 x 轴形成的三角形斜边的角度及其长度。同样，可以使用三角学解决这个问题。可以使用正切函数来计算角度 θ，使用勾股定理来计算距离 r。计算公式如下：

$$r = sqrt(x^2 + y^2)$$
$$\theta = \tan^{-1}(y/x)$$

下面来将位于 QI 中的点(3，4)转换为极坐标，如图 4.6 所示。

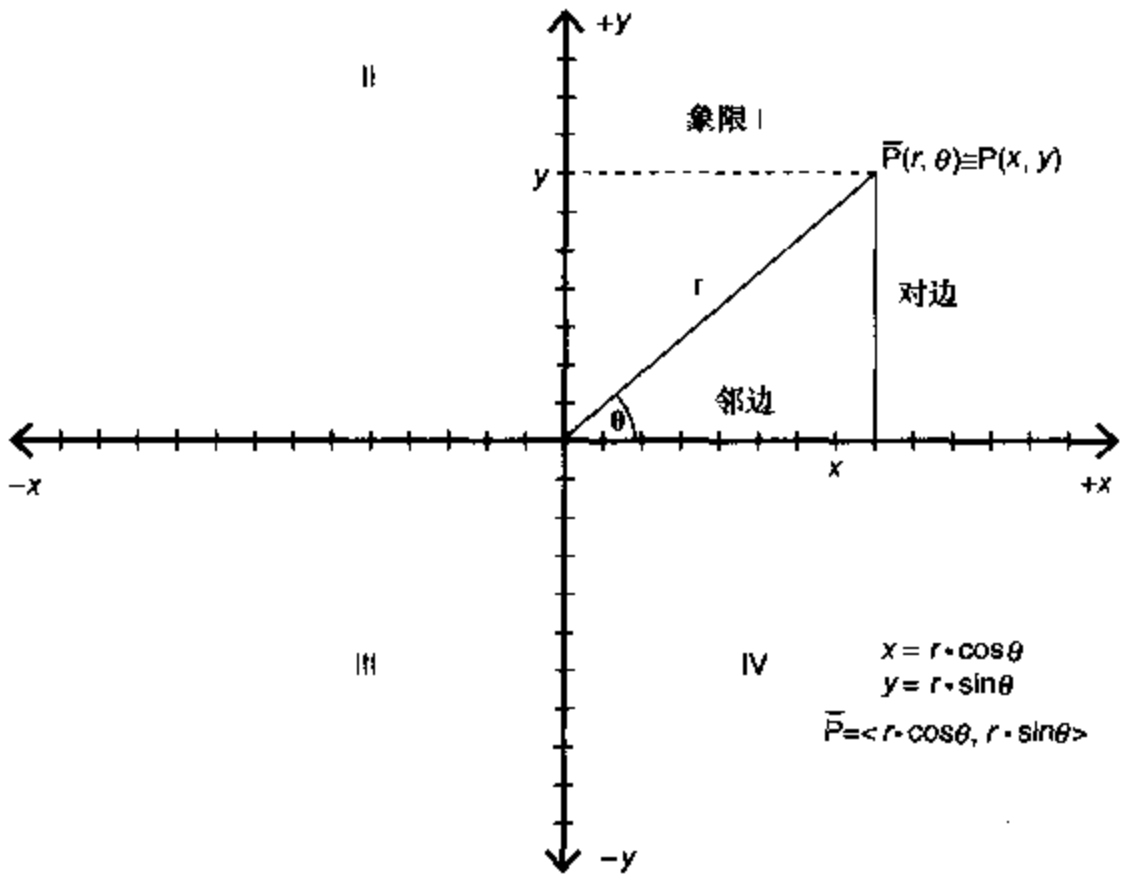


图 4.5　在极坐标和笛卡尔坐标之间进行转换的几何解释

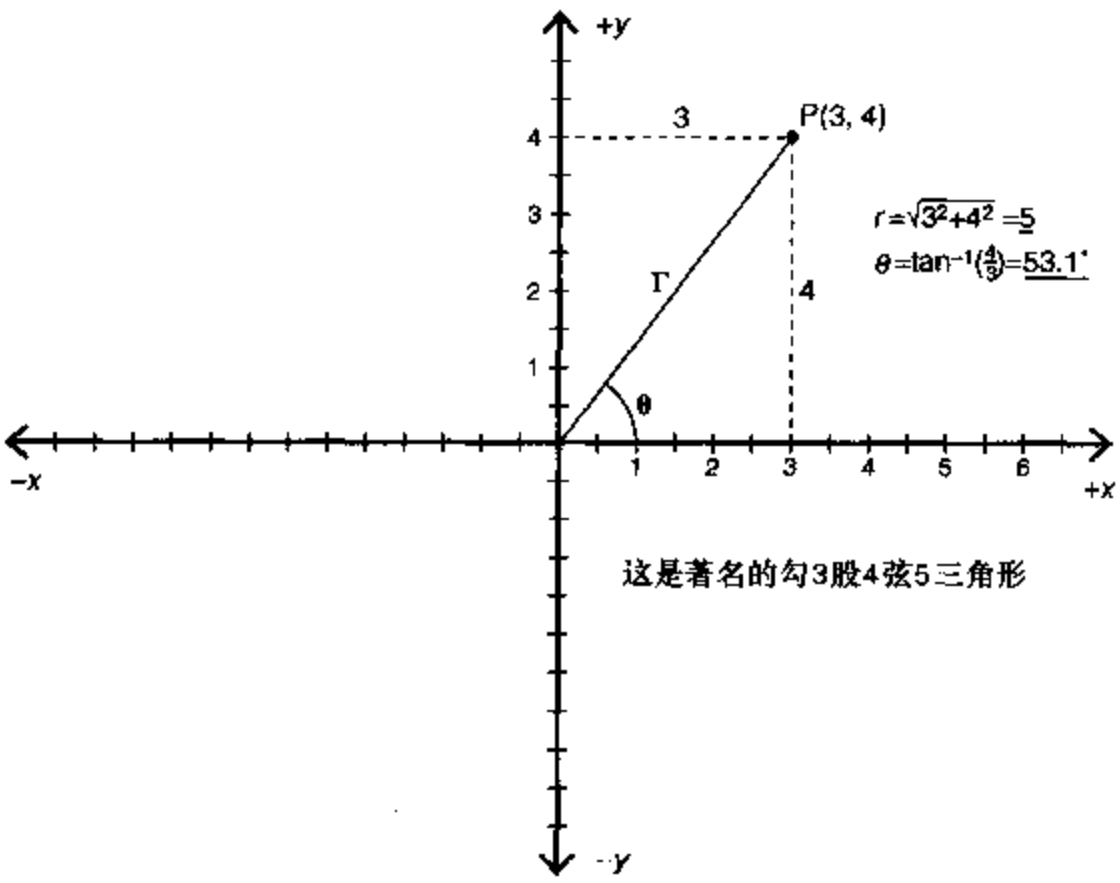


图 4.6　一个将笛卡尔坐标转换为极坐标的范例

将 $x = 3$ 和 $y = 4$ 代入前面的公式，结果如下：

```
r = sqrt(3² + 4²) = 5
θ = tan⁻¹(4/3) = 53.1°
```

请参见图 4.6 的右边。

提示：您可能注意到了，上述公式存在一个问题：当 $x = 0$ 时，y 将变成无穷大！也就是说，当角度 $\theta = 90$ 或 $\theta = 270$ 时，其正切未定义（实际上是无穷大）。因此使用上述公式时，应首先检测 x 是否等于零。$x = 0$ 时，角度为 90 度。

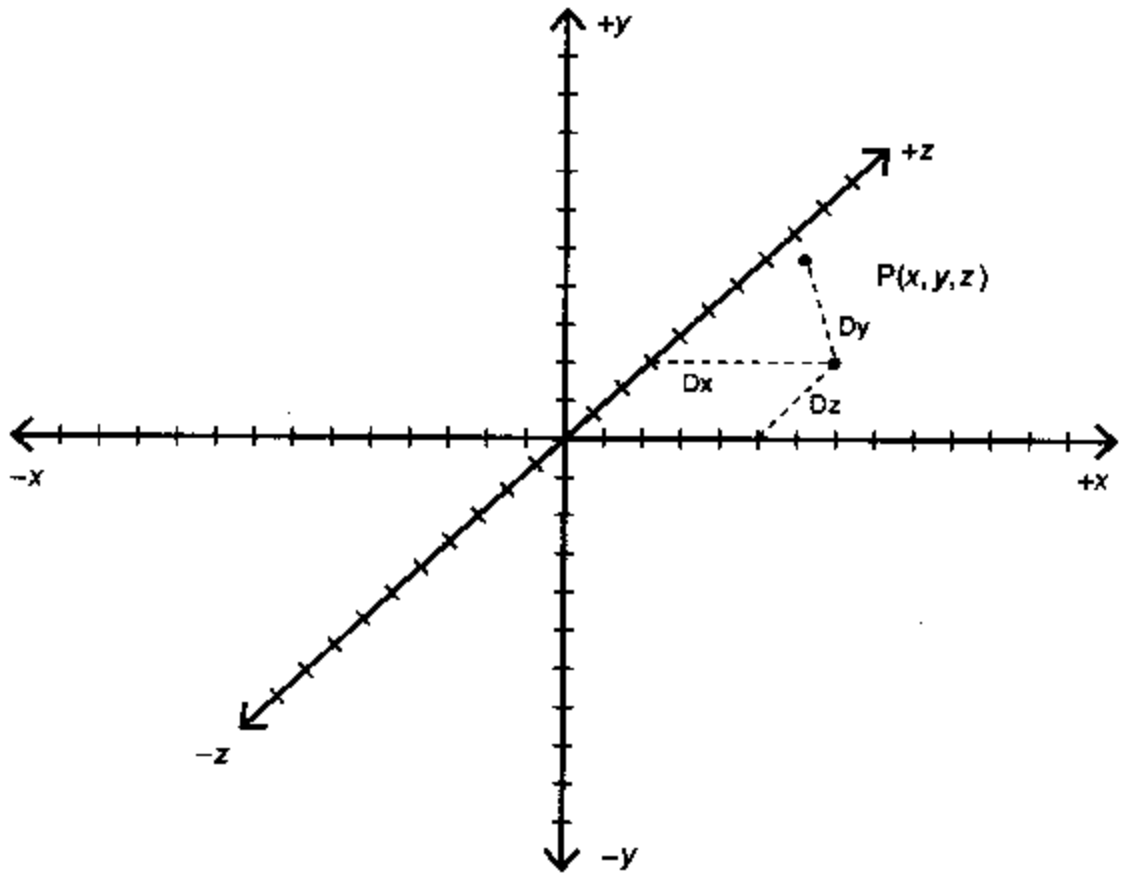
总之，极坐标很有用，理解极坐标以及如何进行坐标变换对于解决很多有关弹道、瞄准和导航的问题非常重要。

4.3　3D 坐标系

读者掌握了 2D 坐标后，接下来介绍 3D 坐标。如果您从事过游戏编程，可能熟悉 3D 坐标系。但这里还是要介绍一下柱面坐标系和球面坐标系，因为它们对实现纹理映射和其他特殊效果很有帮助。先来介绍标准的笛卡尔 3D 坐标系，为介绍柱面坐标系和球面坐标系打好基础。

4.3.1　3D 笛卡尔坐标

3D 笛卡尔坐标系与 2D 笛卡尔坐标系相同，但增加了一条 z 轴。这三条轴互相垂直，坐标系有三个自由度。要定义 3D 空间中的一个点 **p**，需要三个坐标：*x*、*y*、*z*，或简写为 **p**(*x*，*y*，*z*)。另外，这三条轴构成了三个平面：x-y 平面、x-z 平面和 y-z 平面，如图 4.7 所示。这三个平面非常重要，其中每个平面都将空间分成两个区域（半空间）。半空间的概念很重要，后面介绍很多算法时都将用到它。这三个平面将整个空间分成 8 个子空间，因此在 3D 坐标系中，共有 8 个卦限（octans），如图 4.8 所示。



左手坐标系

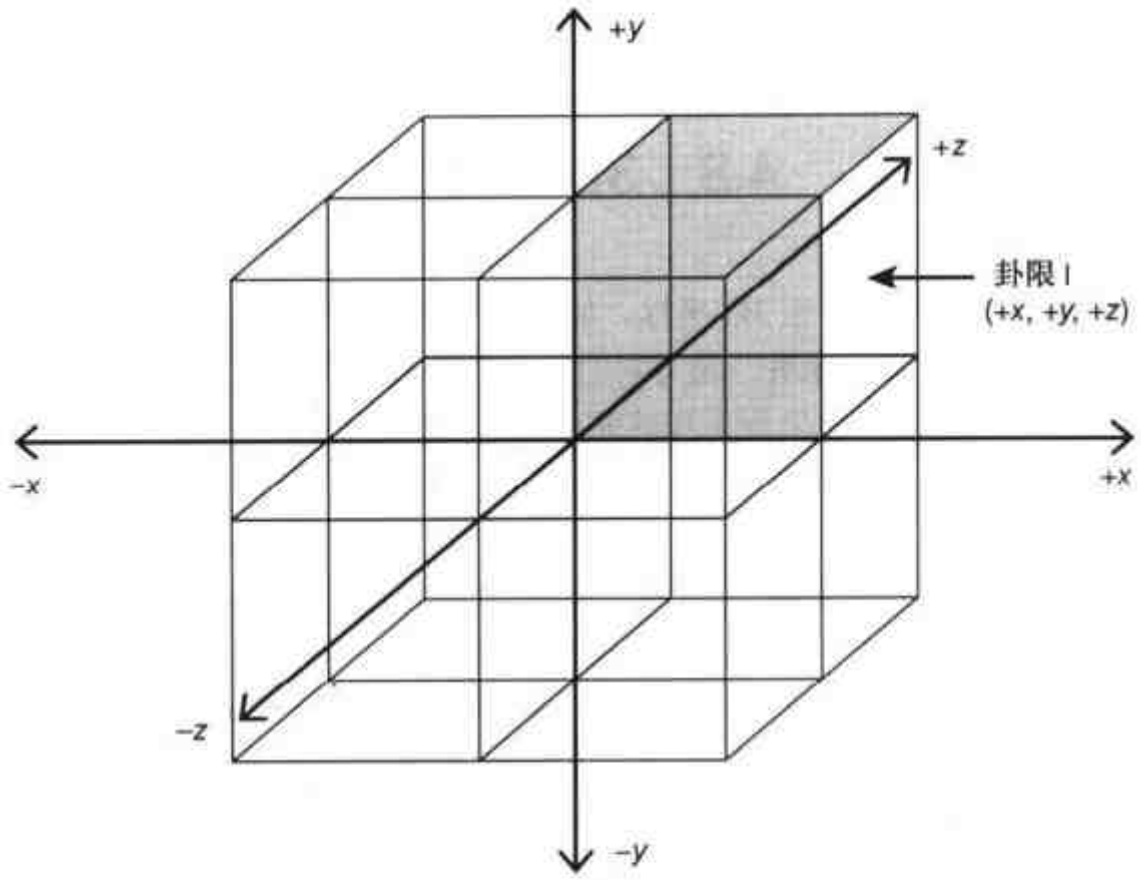
图 4.7　3D 笛卡尔坐标系

新增的 *z* 轴带来了一个小问题：必须决定在两个不同的方向中，哪个代表 *z* 轴的正半空间，哪个代表 *z* 轴的负半空间。因此，有两种不同的 3D 笛卡尔坐标系：左手坐标系和右手坐标系。

提示：将它们命名为左手坐标系和右手坐标系，是因为用左手和右手握住 *z* 轴时，如果其他手指从 *x* 轴环绕到 *y* 轴，则大拇指将指向 z 轴的正方向。

1. 左手坐标系

左手坐标系（LHS）如图 4.9 所示。在 LHS 中，如果将 *x*-*y* 视为纸张上或屏幕上的水平轴和垂直轴，则正 *z* 轴将指向纸内或屏幕内。



左手坐标系

图 4.8　3D 坐标系的卦限

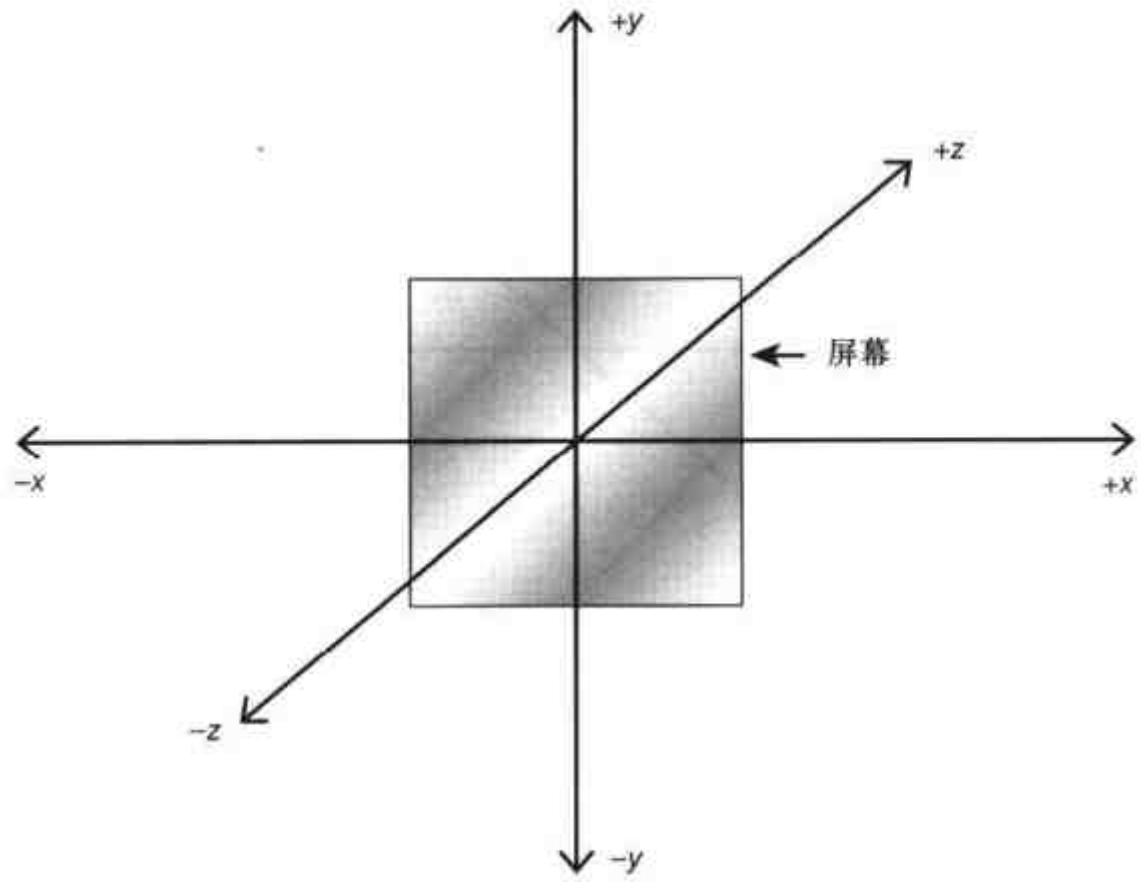


图 4.9　3D 左手坐标系

2. 右手坐标系

右手坐标系（RHS）如图 4.10 所示。在 RHS 中，如果将 *x-y* 平面视为纸面，则正 *z* 轴指向纸外，负 *z* 轴将指向纸内；如果将 *x-y* 视为屏幕，则负 *z* 轴将指向屏幕内，正 *z* 轴将指向屏幕外。

注意：这两种坐标系之间并没有实质上的区别，但本书的 3D 引擎主要使用 LHS。不过，有时也可能使用 RHS，使 3D 图形更易于理解。

本书使用 3D 笛卡尔坐标系，但在解决某些问题时，根据角度和方向来考虑更容易一些，如处理 3D 相

机时。因此，了解另外两种常见的 3D 坐标系是一个不错的主意。

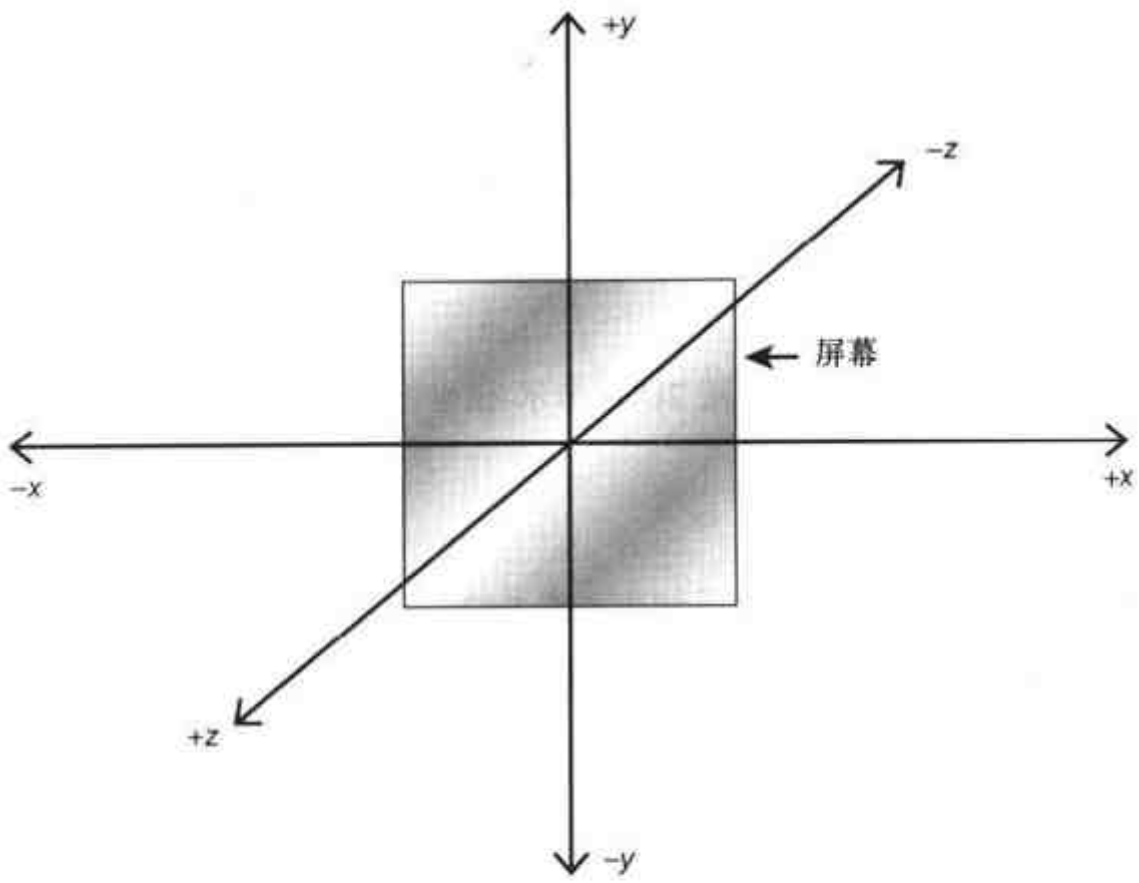


图 4.10　3D 右手坐标系

4.3.2　3D 柱面坐标

有趣的是，在 2D 空间中有极坐标系，但在 3D 空间中有柱面坐标系和球面坐标系，这是因为 3D 空间多了一个自由度。当然，它们之间是可以相互映射的，但每种坐标系都有其用途。3D 柱面坐标系（cylindrical coordinate systm）最接近于 2D 极坐标系，因为它只是在 2D 极坐标系的基础上增加了一条 z 轴，如图 4.11 所示。

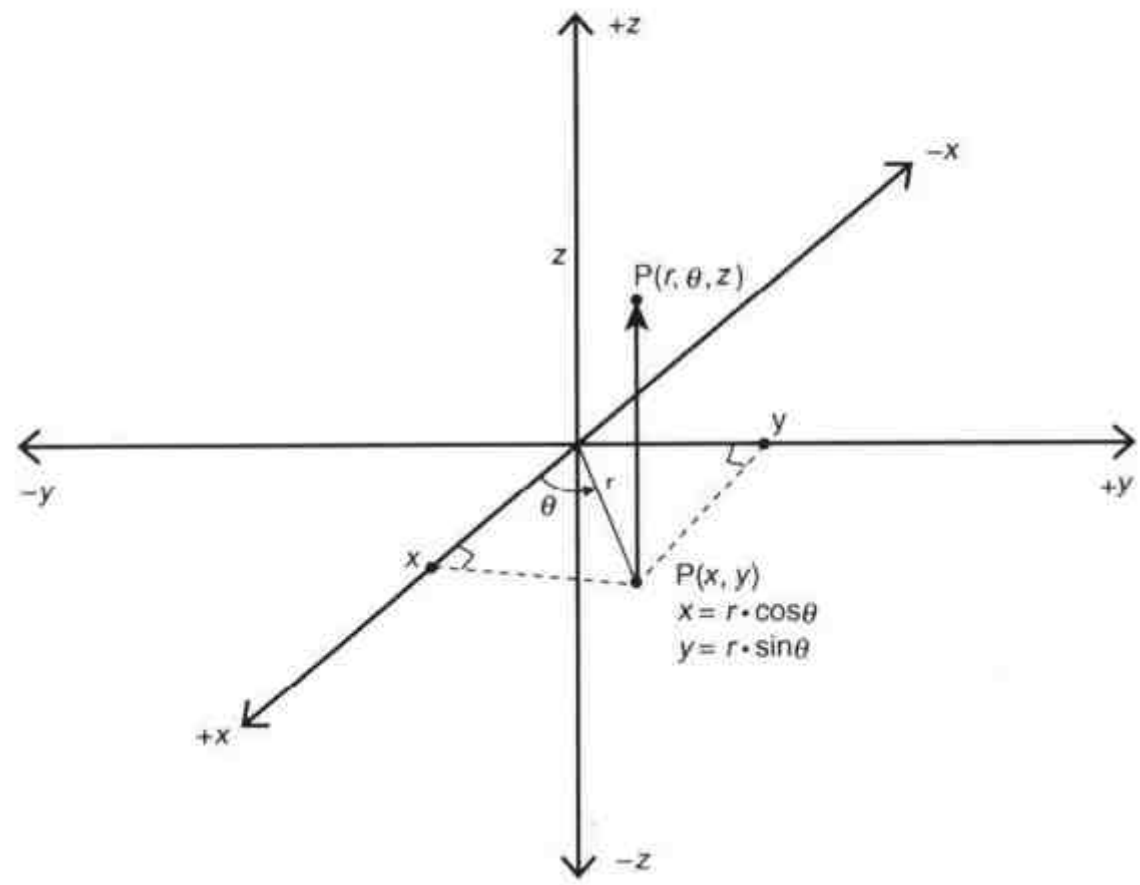


图 4.11　柱面坐标系

图 4.11 说明了标准的柱面坐标系。$z=0$ 的 x-y 平面是一个标准 2D 极坐标系，在柱面坐标系中定义 3D 点时，首先在 $z=0$ 的 x-y 平面上定义一个 2D 点 $\mathbf{p}(r, \theta)$，然后使用 z 坐标定义该点应平行于 z 轴“爬”多远。为便于说明，图中使用的是右手坐标系。

在 3D 笛卡尔坐标和极坐标之间进行转换

在 3D 笛卡尔坐标和柱面坐标之间进行转换很容易，只需使用 2D 转换公式，并保留 z 不变即可：

```
x = r*cos(θ)
y = r*sin(θ)
z = z
```

将笛卡尔坐标转换为柱面坐标时，也使用 2D 笛卡尔-极坐标转换公式，并保留 z 坐标不变：

```
r = sqrt(x² + y²)
θ = tan⁻¹(y/x)
z = z
```

警告：当 $\theta=90$ 和 $\theta=270$ 时，也存在反正切为无穷大的问题。

对于很多问题，使用柱面坐标是非常方便的，如在第一人称射击游戏中控制相机、环境映射等。

4.3.3 3D 球面坐标

3D 球面坐标系比其他所有坐标系都复杂些。在 3D 球面坐标系中，使用两个角度和到原点的距离来定义点。图 4.12 是一个球面坐标系。要在球面坐标系中定义一个点，需要两个角度和到原点的距离，通常表示为 $\mathbf{p}(\rho, \phi, \theta)$，其中：

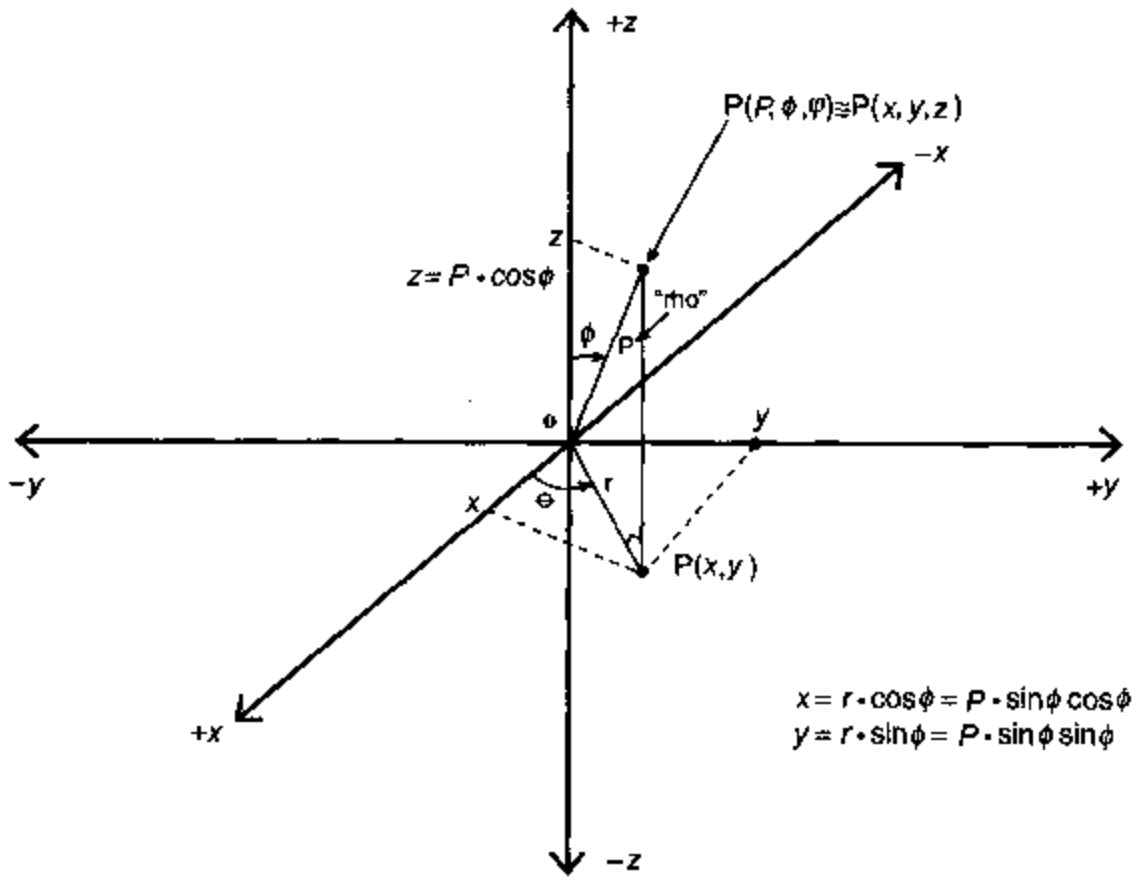


图 4.12　球面坐标及其转换

- ρ是点 **p** 到原点的距离。
- φ是原点(**o**)到点 **p** 的线段与正 **z** 轴（这里使用的是右手坐标系）之间的夹角。
- θ是原点 **o** 到点 **p** 的线段在 x-y 平面上的投影与正 x 轴之间的夹角，它正好是标准 2D 极角 θ。另外，$0 <= \theta <= 2\pi$。

因此，同样可以使用 2D 极坐标的知识，来推导出球面坐标。下面介绍如何在 3D 笛卡尔坐标与球面坐标之间进行转换。

首先，由于使用文字来解释这种转换比较困难，因此请仔细看一下图 4.12。要将球面坐标转换为 3D

笛卡尔坐标(*x*，*y*，*z*)，可以分两步来完成。首先，将线段 **o->p** 投影到 *x-y* 平面上，以消除 *z* 坐标。这样，问题就变成 2D 的，而对于这种问题，我们已经有解决方案。然后，再回过头来，根据(*ρ*，*ϕ*，*θ*)计算出 *z* 值。具体步骤如下：

从线段 **o->p** 到 *x-y* 平面的投影可知：

```
r = ρ*sin(ϕ)
z = ρ*cos(ϕ)
```

在 *x-y* 平面上：

```
x = r*cos(θ)
y = r*sin(θ)
```

将 *r* 代入到上面计算 *x*、*y* 的公式中，结果如下：

```
x = ρ*sin(ϕ)*cos(θ)
y = ρ*sin(ϕ)*sin(θ)
z = ρ*cos(ϕ)
```

下面介绍如何将 3D 迪卡尔坐标转换为球面坐标。

由于 $x^2+y^2=\rho^2$，同时 $x^2+y^2=r^2$，因此：

```
r = sqrt(x² + y²)
ρ = sqrt(x² + y² + z²)
θ = tan⁻¹(y/x)
```

可以根据下述 *r* 和*ρ*之间的关系：

```
r = ρ*sin(ϕ)
```

来求解*ϕ*：

```
ϕ = sin⁻¹(r/ρ)
```

也可以根据下述关系：

```
z = ρ*cos(ϕ)
```

来求解*ϕ*：

```
ϕ = cos⁻¹(z/ρ)
```

4.4　三　角　学

三角学是一门研究角度、形状及其关系的科学。大多数三角学都以对直角三角形的分析为基础，如图 4.13 所示。直角三角形有三个内角和三条边。与直角相对的角通常用作参考点，有时也叫作底角（base angle）。根据它可以对三角形进行标记：与底角相邻的边叫作邻边，与底角相对的边叫作对边，连接它们的边叫作斜边。不要低估了三角形和相似三角形的作用，我很少见到有哪种图形算法不使用三角形或相似三角形的。虽然有关三角形的知识很容易，但熟练它们对图形编程来说至关重要。

4.4.1　直角三角形

下面来复习一下有关三角形的一些事实。

注意：本书将交替使用 PI 和符号π，但，*PI* 通常是头文件 MATH.H 中的计算机常量，而π则意味着 3.14159……

事实 1　整个圆周为 360 度（2π弧度），因此 180 度等于π弧度。计算机函数 sin() 和 cos() 的参数以弧度为单位，而不是以度为单位！表 4.4 列出了这些值。

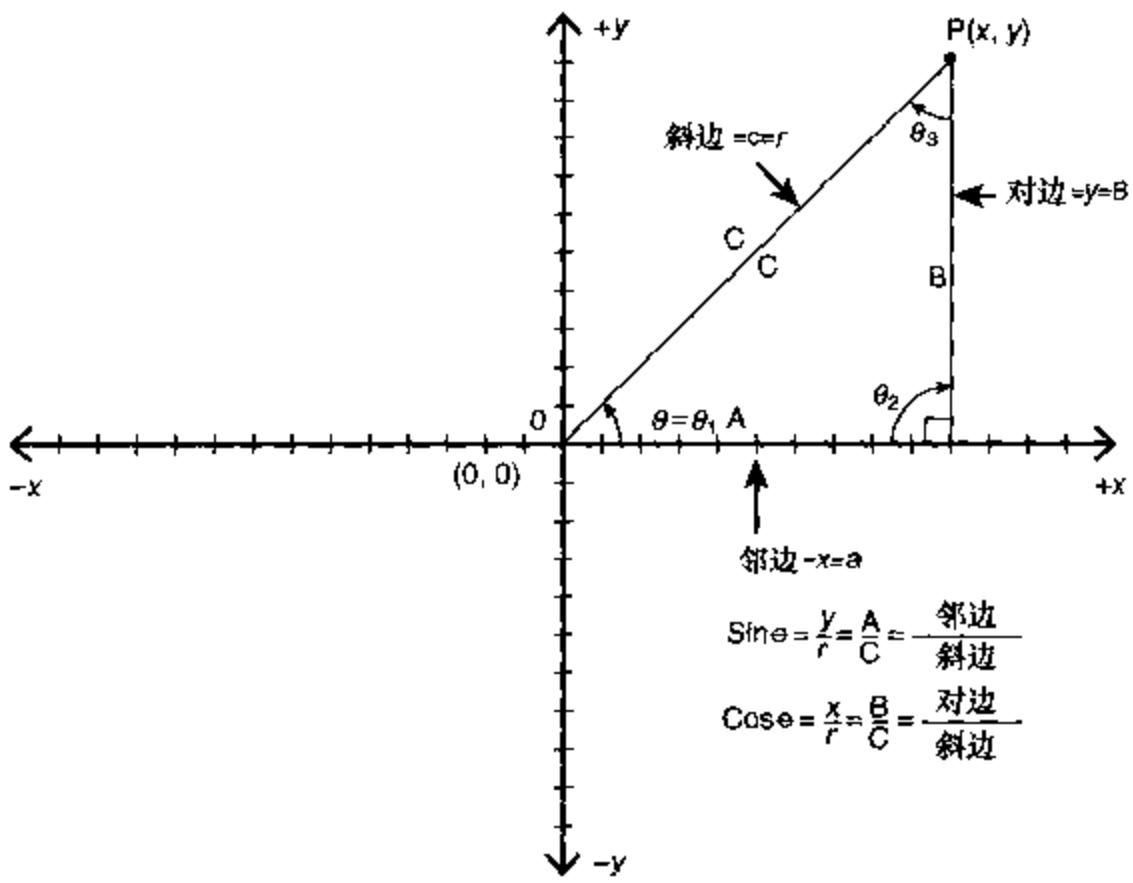


图 4.13　直角三角形

表 4.4　　**弧度与度之间的关系**

以度为单位的角度	以弧度为单位的角度	以度为单位的角度	以弧度为单位的角度
360 度	2*PI 弧度（大约 6.28 弧度）	57.296 度	1 弧度
180 度	PI 弧度（大约 3.14159 弧度）	1.0 度	0.0175 弧度

事实 2　三角形的三个内角之和为 180 度（PI 弧度）。

事实 3　参见图 4.13 中的直角三角形，与角 θ_1 相对的边叫作对边，它下面的边叫作邻边，而那条长边叫作斜边。

事实 4　直角三角形的对边和邻边的平方和等于斜边的平方，这被称为勾股定理。因此，知道直角三角形中两条边的长度后，便可以计算出第三条边的长度。

事实 5　数学家喜欢使用的三个主要的三角函数正弦、余弦和正切。它们的定义如下：

$$\cos(\theta)=\frac{\text{邻边}}{\text{斜边}}=\frac{x}{r}$$

其中定义域为：0 <= θ <= 2*PI，值域为-1 到 1。

$$\sin(\theta)=\frac{\text{对边}}{\text{斜边}}=\frac{y}{r}$$

其中定义域为：0 <= θ <= 2*PI；值域为-1 到 1。

$$\tan(\theta)=\frac{\sin(\theta)}{\cos(\theta)}=\frac{\text{对边}/\text{斜边}}{\text{邻边}/\text{斜边}}$$
$$=\frac{\text{对边}}{\text{邻边}}=\frac{y}{x}=\text{斜率}=M$$

其中定义域为：-PI/2 <= θ <= PI/2，值域为负无穷大到正无穷大。

图 4.14 是这些三角函数的图形。这些函数都是周期函数，其中正弦函数和余弦函数的周期都是 2*PI，而正切函数的周期为 PI。当 θ 除以 PI 的余数为 PI/2 时，*tan*(θ) 将为负无穷大或正无穷大。

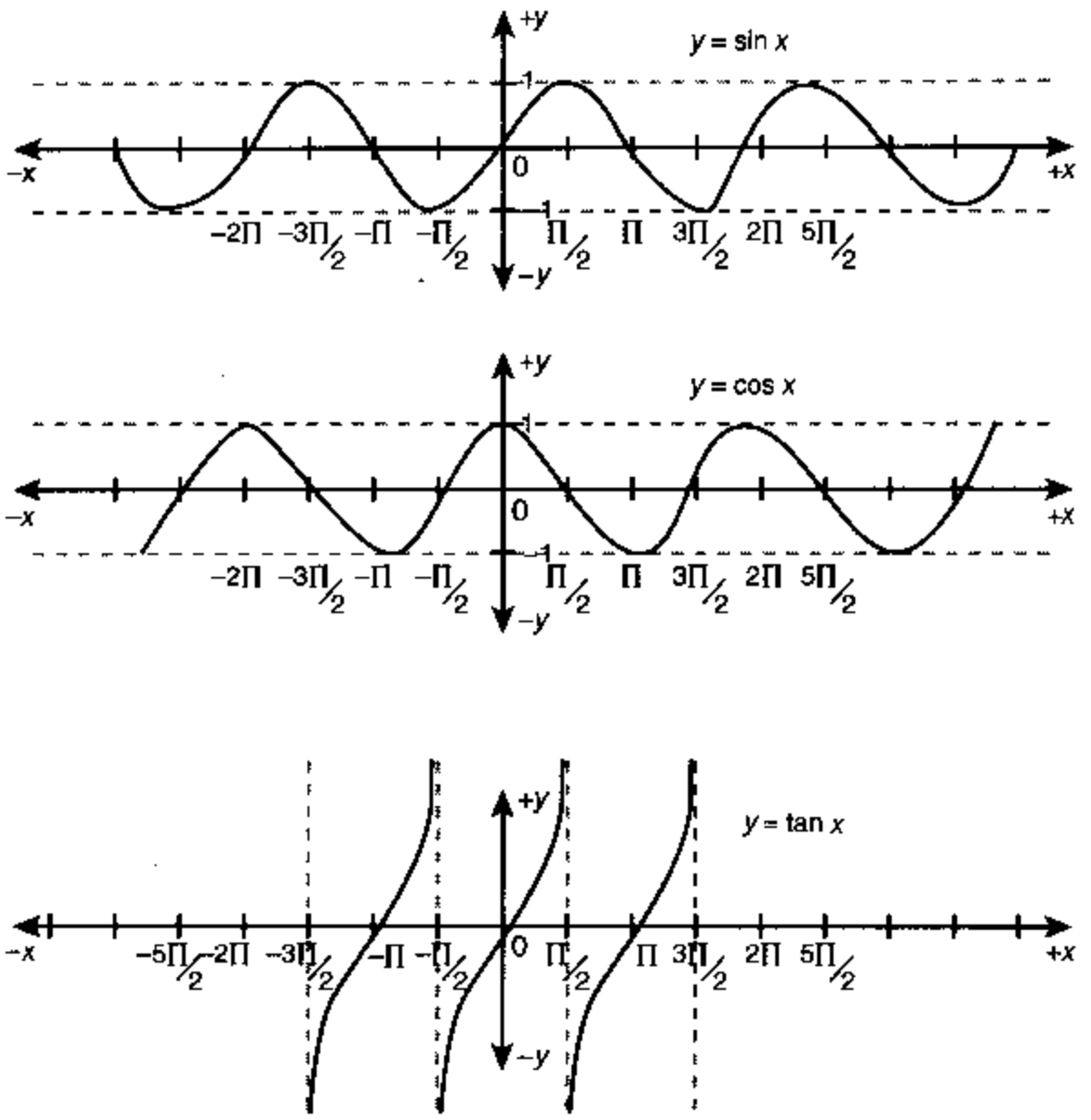


图 4.14　基本三角函数

4.4.2　反三角函数

很多时候当您使用三角函数，如正弦、余弦和正切函数时，需要根据几何学知识创建一个公式，使用函数值反算出角度。也就是说，根据正弦、余弦或正切值反算出对应的角度。这种函数叫反函数，使用上标“-1”表示：

$\theta = \cos^{-1}(x)$，其中 $0 <= \theta <= 2*PI$，$-1 <= x <= 1$
$\theta = \sin^{-1}(x)$，其中 $0 <= \theta <= 2*PI$，$-1 <= x <= 1$
$\theta = \tan^{-1}(x)$，其中 $-PI/2 <= \theta <= PI/2$，$-\infty < x < +\infty$

注意：在计算机上，无法表示无穷大，但在这种环境下，无穷大指的是这样的数字，即它们足够大，比我们考虑的数字空间中的任何数还要大很多倍。例如，如果游戏空间为 1000×1000 的网格，则可以认为 100 000 000 接近于无穷大。

使用上述反函数相当容易：用一个数值代替 x，就可以计算出角度 θ（这种工作将由计算机完成）。然而，我们面临的困境是，计算正弦、余弦和正切函数的速度很慢（即使是在计算机上），而计算反函数的速度还要慢 10 倍以上！因此，应确认是否确实需要知道角度。很多时候，只需在算法中使用 x 值即可。当然，如果确实需要角度，可以使用一些技巧，如查找表和插值。

4.4.3　三角恒等式

三角恒等式和技巧数不胜数，要全部证明它们，需要整本书的篇幅，这里只列出游戏程序员应该知道的一些恒等式：

倒数函数：

正割：$\csc(\theta) = 1/\sin(\theta)$
余割：$\sec(\theta) = 1/\cos(\theta)$

余切：cot(θ) = 1/tan(θ)

勾股定理的三角函数表示：

$\sin(\theta)^2+\cos(\theta)^2 = 1$

转换恒等式：

```
sin(θ) = cos(θ-PI/2)
```

反射恒等式：

```
sin(-θ) = -sin(θ)
cos(-θ) = cos(θ)
```

角度相加恒等式：

$$\sin(\theta_1 + \theta_2) = \sin(\theta_1)*\cos(\theta_2)+\cos(\theta_1)*\sin(\theta_2)$$
$$\cos(\theta_1 + \theta_2) = \cos(\theta_1)*\cos(\theta_2)-\sin(\theta_1)*\sin(\theta_2)$$
$$\sin(\theta_1 - \theta_2) = \sin(\theta_1)*\cos(\theta_2)-\cos(\theta_1)*\sin(\theta_2)$$
$$\cos(\theta_1 - \theta_2) = \cos(\theta_1)*\cos(\theta_2)+\sin(\theta_1)*\sin(\theta_2)$$

当然，可以根据它们推导出很多其他三角恒等式。一般来说，三角恒等式可帮助简化复杂的三角公式，避免执行数学运算。因此，得到基于正弦、余弦或正切的公式后，务必参考三角学方面的书籍，看能否对公式进行化简，进而减少计算量。请切记，速度是游戏的关键！

技巧：后面讨论矩阵变换时，可能将多种操作（如平移、旋转、缩放等）组合起来。旋转操作本身可能是多个用正弦和余弦表示的旋转变换的组合。因此，计算矩阵乘法时，可以使用三角恒等式来简化矩阵，编写一个具体的函数来执行变换操作，而不是盲目地使用通用的矩阵乘法。

4.5 向　　量

向量是所有 3D 算法的基础，因此对于游戏程序员来说非常重要。向量由多个分量组成，2D/3D 向量表示一条有向线段。向量是用起点和终点定义的，如图 4.15 所示，其中向量 **u** 是用两个点 **p1**（起点）和 **p2**（终点）定义的。向量 **u** = $\langle u_x, u_y \rangle$从 **p1**(x1，y1)指向 **p2**(x2，y2)。要计算向量 **u**，只需将终点和起点相减：

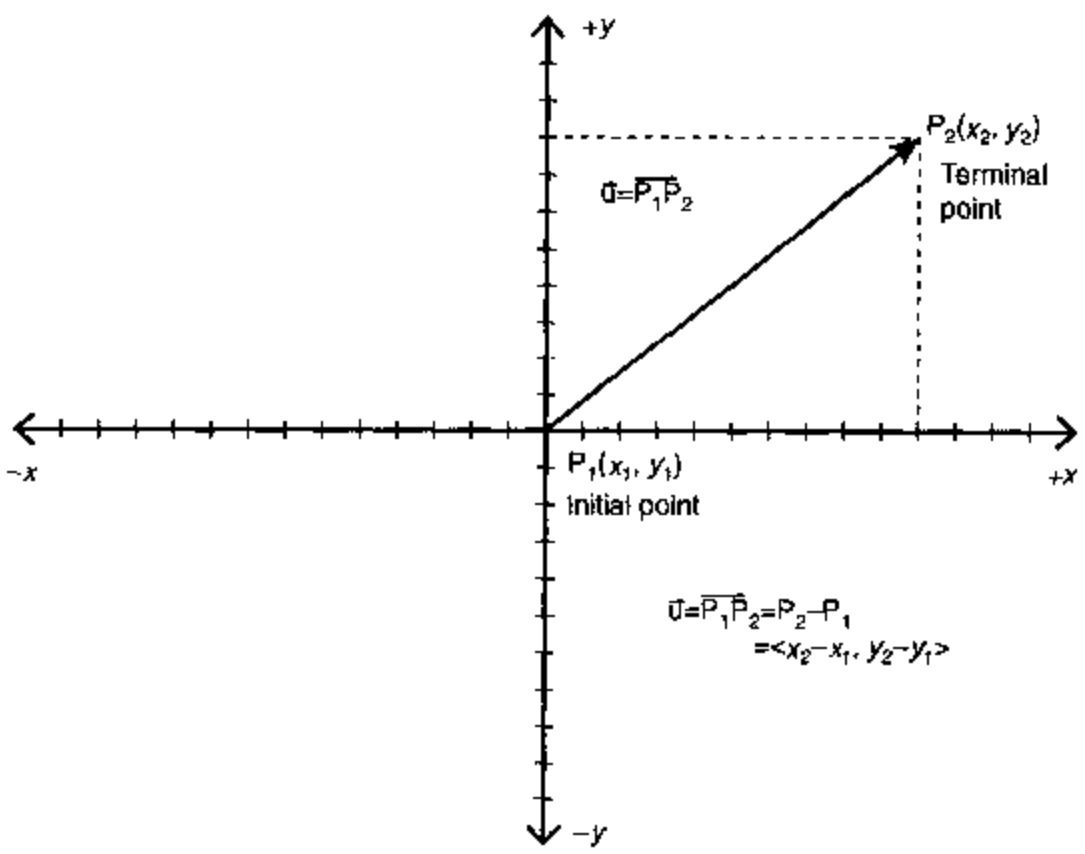


图 4.15　向量的定义

$$\mathbf{u} = \mathbf{p2} - \mathbf{p1} = (x2-x1,\ y2-y1) = \langle u_x,\ u_y \rangle$$

我们通常使用粗体字母来表示向量，如 **u**，并将分量用尖括号括起，如$\langle u_x,\ u_y \rangle$。

向量是从一个点到另一点的有向线段，但该线段可以表示很多概念，如速度、加速度等。需要注意的是，向量被定义后，总是相对于原点的。这意味着当您创建了一个从点 **p1** 到点 **p2** 的向量后，该向量的起点总是(0，0)（在 3D 空间中为（0，0，0））。2D 向量由两个数组成，而 3D 向量由三个数组成，因此只定义了 2D 或 3D 空间中的终点，起点总是为原点。这并不意味着不能平移向量并使用它们来执行各种几何操作；而只意味着您必须记住向量实际上是什么。

向量优点在于可以对它们执行的操作。向量实际上是一个有序数集，可以通过分别对每个分量执行数学运算，来对向量执行标准的数学运算。

注意：向量可以由任意个分量组成。在计算机图形学中，处理的是 2D 和 3D 向量；即向量的形式为 $a = \langle x, y \rangle$和 $b = \langle x, y, z \rangle$。$n$ 维向量的形式为：

$$\mathbf{c} = \langle c1,\ c2,\ c3,\ \ldots,\ cn \rangle$$

n 维向量用于代表变量集而不是几何空间，因为超过 3 维后便是超空间，正常空间只有三维。

4.5.1　向量长度

使用向量时，经常遇到的一个问题是如何计算其长度。向量的长度被称为范数（norm），本书通过在向量两边分别添加竖线来表示，如|**u**|表示向量 **u** 的长度。向量长度是从原点到向量表示的终点的距离，具体地说是各个分量平方和的平方根。因此，可以使用勾股定理来计算 2D 和 3D 向量的长度。在 2D 和 3D 空间中，计算|**u**|的公式如下

$$|\mathbf{u}| = \mathrm{sqrt}(u_x^2 + u_y^2)$$

$$|\mathbf{u}| = \mathrm{sqrt}(u_x^2 + u_y^2 + u_z^2)$$

4.5.2　归一化

知道向量的长度后，就可以对其进行归一化（normalize），即进行缩放，使其长度为 1.0，同时方向保持不变。像标量 1.0 一样，单位向量也有很多不错的性质。例如，有时候，我们可能执行很多运算，且只关心结果向量的方向，而不关心其长度，也就是说只需要一个与结果向量的方向相同，长度为 1.0 的向量。归一化可以提供这种向量。给定向量 $n = \langle nx,\ ny \rangle$，通常这样表示其归一化版本：

$$\hat{\mathbf{n}}$$

但为便于排版，我们将使用斜体小写字母（或加上一个撇号）来表示：n 或 n'。

计算向量 n 的归一化版本的公式如下：

$$n' = n/|n|$$

非常简单。归一化版本为向量除以其长度。

4.5.3　向量和标量的乘法

一种向量运算是缩放。例如，假设有一个表示速度的向量，要提高或降低速度，可以使用缩放运算。缩放是通过将每个分量乘以一个标量来完成的，例如：

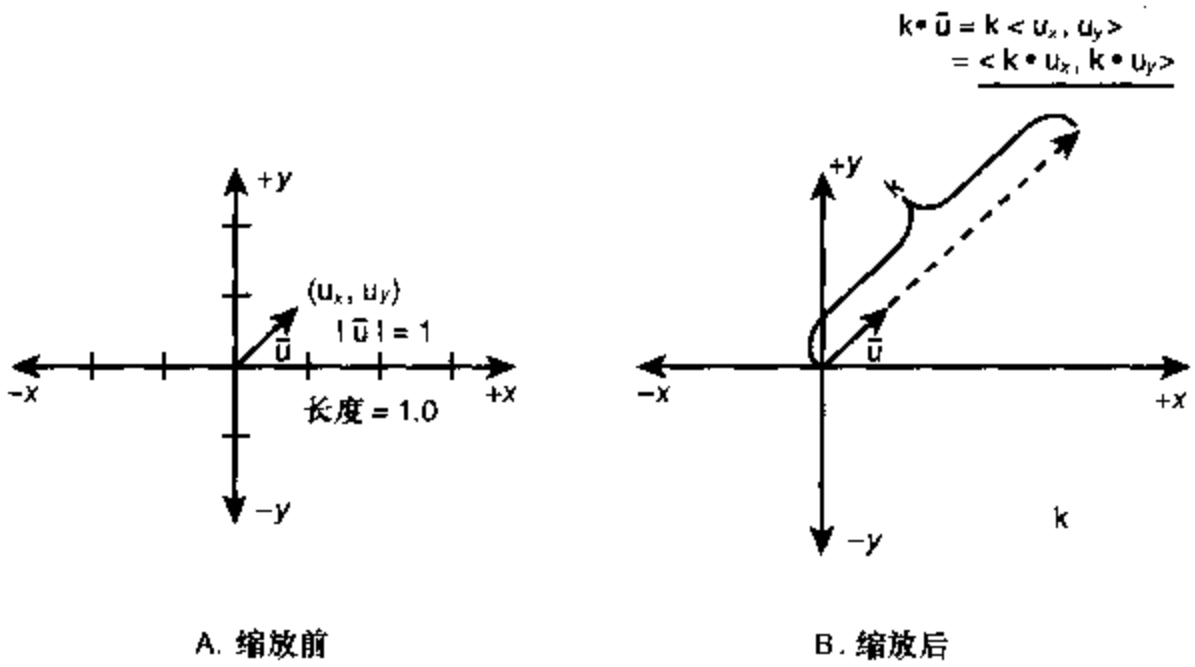
令 $\mathbf{u} = \langle u_x,\ u_y \rangle$，$k$ 为实数常量，则：

$$k*\mathbf{u} = k*\langle u_x, u_y \rangle = \langle k*u_x, k*u_y \rangle$$

图 4.16 图示缩放运算。要反转向量的方向，可以将其乘以–1，如图 4.17 所示。用数学语言说，情况如下：

假设 $\mathbf{u} = \langle u_x,\ u_y \rangle$，则方向与 **u** 相反的向量为

$$-1*\mathbf{u} = -1* \langle u_x, u_y \rangle = \langle -u_x, -u_y \rangle$$



A. 缩放前　　B. 缩放后

图 4.16　向量缩放

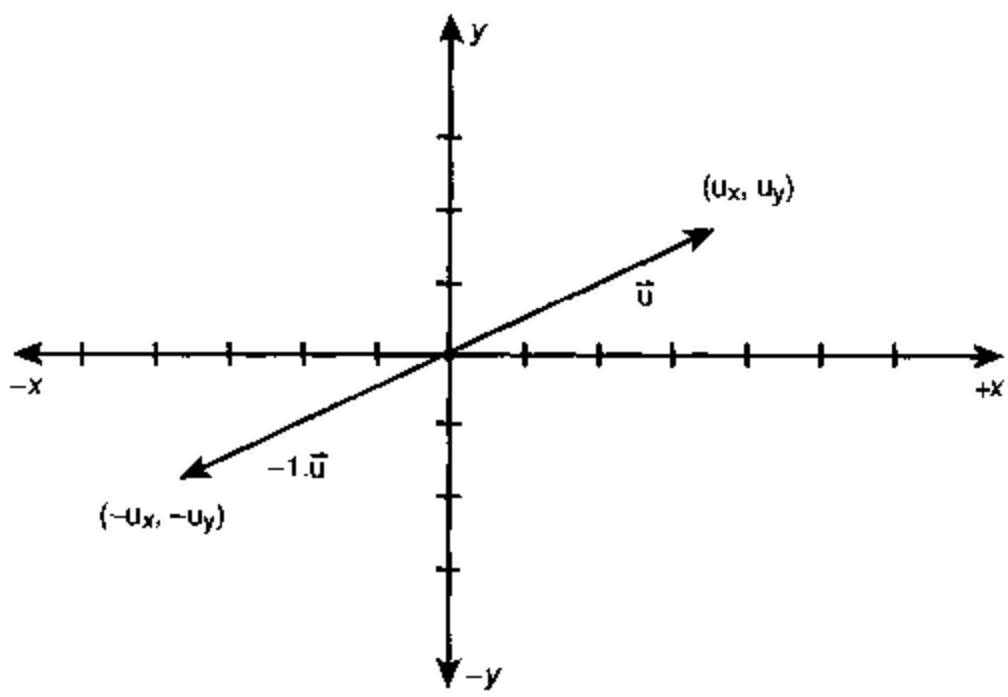


图 4.17　反转向量的方向

4.5.4　向量加法

要将多个向量相加，只需将各个分量分别相加即可。图 4.18 图示了 2D 向量的加法运算：将向量 u 和 v 相加，结果为向量 r。请注意向量加法的几何意义：平行移动向量 v，使其起点与向量 u 的终点重合，然后画出三角形的另一条边。从几何学上说，这种操作相当于下面的数学运算：

$$\mathbf{u} + \mathbf{v} = \langle u_x, u_y \rangle + \langle v_x, v_y \rangle = \langle u_x + v_x, u_y + v_y \rangle$$

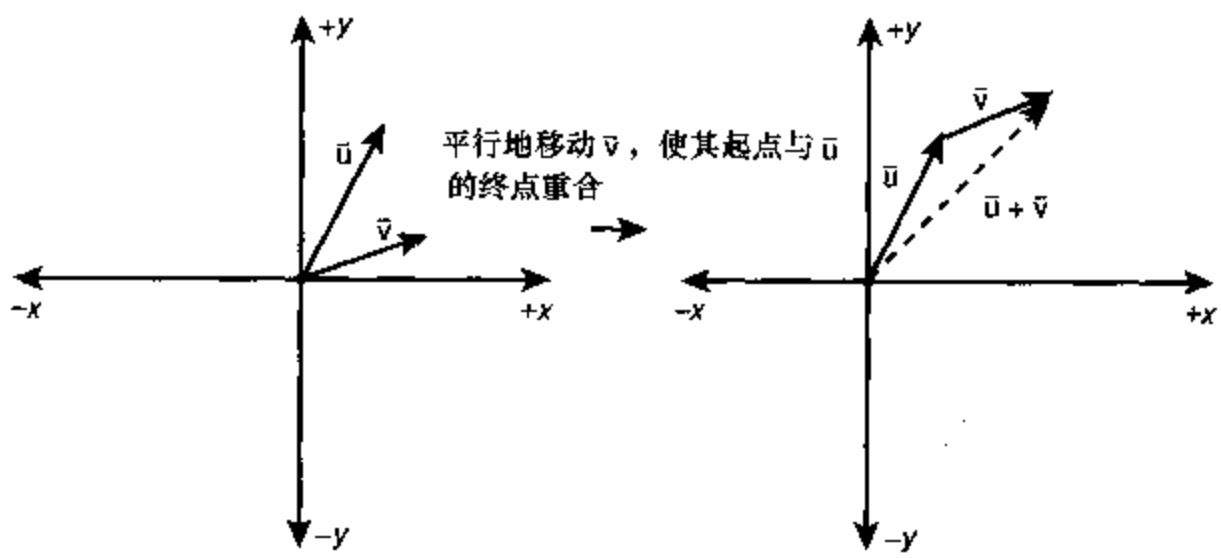


图 4.18　向量加法

因此，要在图纸上将任意多个向量相加，可以将这些向量“首尾相连”，从第一个向量的起点到最后一个向量的终点的线段，就是这些向量的和。

4.5.5　向量减法

向量减法实际上是加上一个方向相反的向量；然而，有时候以图形方式来表示向量减法更直观。图 4.19 图示了 $u-v$。$u-v$ 是从 v 到 u 的向量，而 $v-u$ 是 u（起点）到 v（终点）的向量。用数学语言说，情况如下：

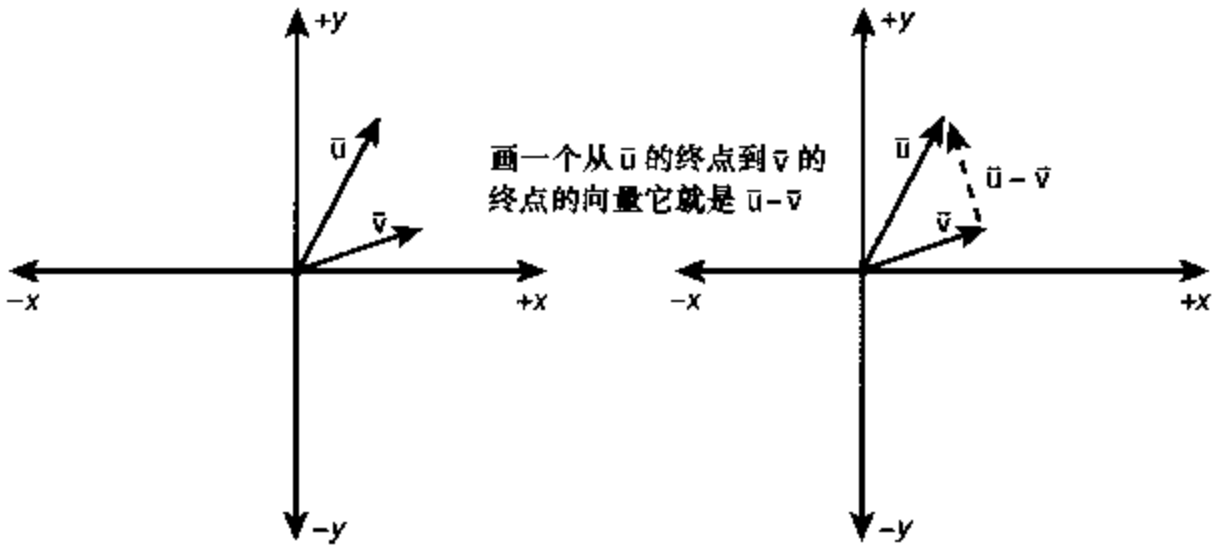


图 4.19　向量减法

$$u - v = \langle u_x, u_y \rangle - \langle v_x, v_y \rangle = \langle u_x - v_x, u_y - v_y \rangle$$

这可能更容易记住，但有时候在图纸上进行计算更好些，因为可以直观地看到计算结果。知道如何在图纸上执行向量加法和减法，对于编写渲染算法很有帮助。

4.5.6　点积

读者可能会问，“可以将两个向量相乘吗？”答案是肯定的，但直接将向量的各个分量相乘并不是很有用：

$$u \otimes v = \langle u_x * v_x, u_y * v_y \rangle$$

另一种被称为点积的乘法非常有用，其定义如下：

$$u \cdot v = u_x * v_x + u_y * v_y$$

点积通常用点(.)表示，它将各个分量分别相乘后相加，得到一个标量，而不是将各个分量相乘，并保留向量形式。读者可能会问，点积有什么用呢？其结果不再是向量了！但点积相当于下列表达式：

$$u \cdot v = |u| * |v| * \cos(\theta)$$

该表达式指出，u 和 v 的点积等于向量 u 的长度乘以向量 v 的长度，再乘以它们的夹角的余弦。组合上述两个表达式可得到如下结果：

$$u \cdot v = u_x * v_x + u_y * v_y$$
$$u \cdot v = |u| * |v| * \cos(\theta)$$
$$u_x * v_x + u_y * v_y = |u| * |v| * \cos\theta$$

这是一个很有趣的公式，它提供了一种计算两个向量之间夹角的方法，如图 4.20 所示，因此点积是一种很有用的运算。如果读者还不明白，可以对上述公式两边求反余弦，结果如下：

$$\theta = \cos^{-1}(u_x * v_x + u_y * v_y / |u| * |v|)$$

将 $u.v = (ux * vx + uy * vy)$ 代入上述公式，结果如下：

$$\theta = \cos^{-1}(u.v / |u| * |v|)$$

这是一个功能非常强大的工具，也是很多 3D 图形学算法的基础。如果 u 和 v 都是单元向量，即 u 和 v 的长度都是 1.0，则 $|u| * |v| = 1.0$，上述公式可进一步简化为：

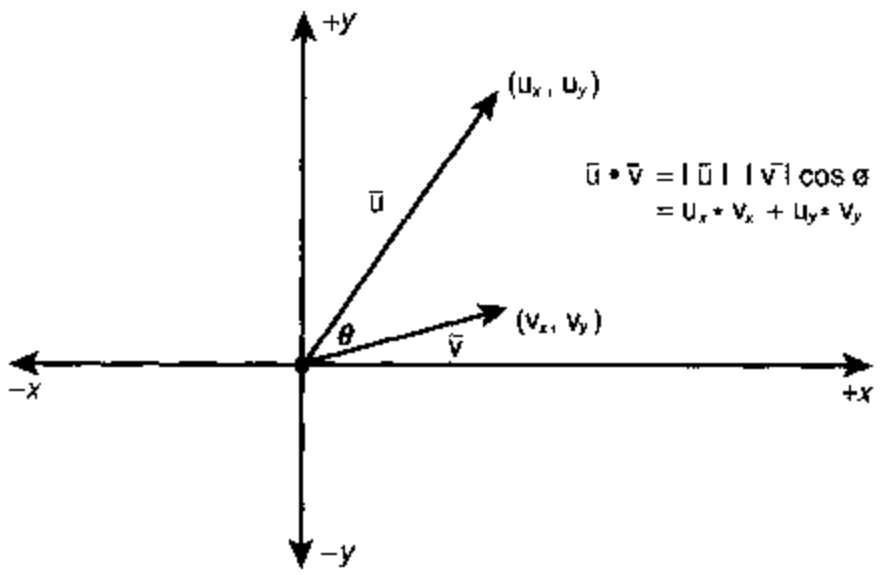


图 4.20　点积

$\theta = \cos^{-1}(\mathbf{u}.\mathbf{v})$, for $|\mathbf{u}|=|\mathbf{v}| = 1.0$

下面是一些关于点积的有用事实：

事实 1　如果向量 u 和 v 之间的夹角为 90 度（互相垂直），则 $u.v=0$。

事实 2　如果向量 u 和 v 之间的夹角小于 90 度（锐角），则 $u.v>0$。

事实 3　如果向量 u 和 v 之间的夹角大于 90 度（钝角），则 $u.v<0$。

事实 4　如果向量 u 和 v 相等，则 $u.v=|u|^2=|v|^2$。

图 4.21 图示了这些事实。

另外，点积还可用于很多其他计算。在计算机图形学和游戏编程中，点积的重要用途之一是计算向量在给定方向上的分量(投影)，如图 4.22 所示。例如，假设有一个向量 v，它代表游戏某个角色的运动轨道；还有一个向量 u，它表示游戏中另一个角色的运动轨道。很多情况下，需要知道 u 在 v 方向的分量，即向量 u 在向量 v 上的投影（$Proj_v u$）。可以使用点积来完成这种计算。

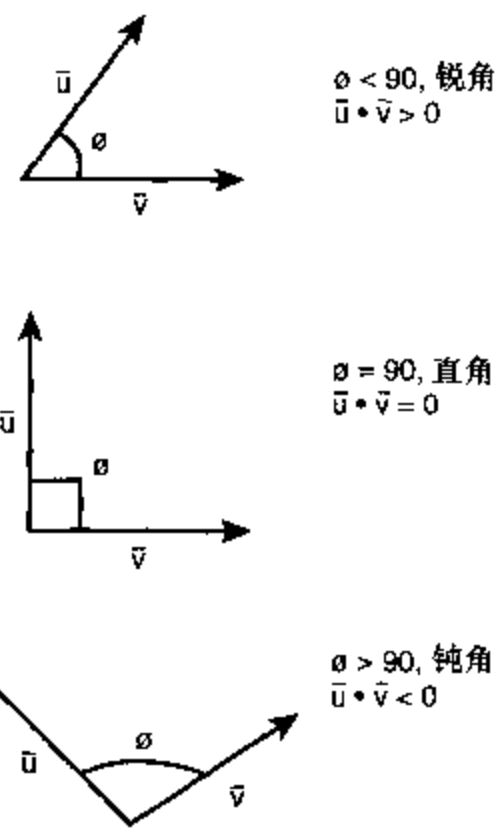


图 4.21　向量的夹角及其与点积的关系

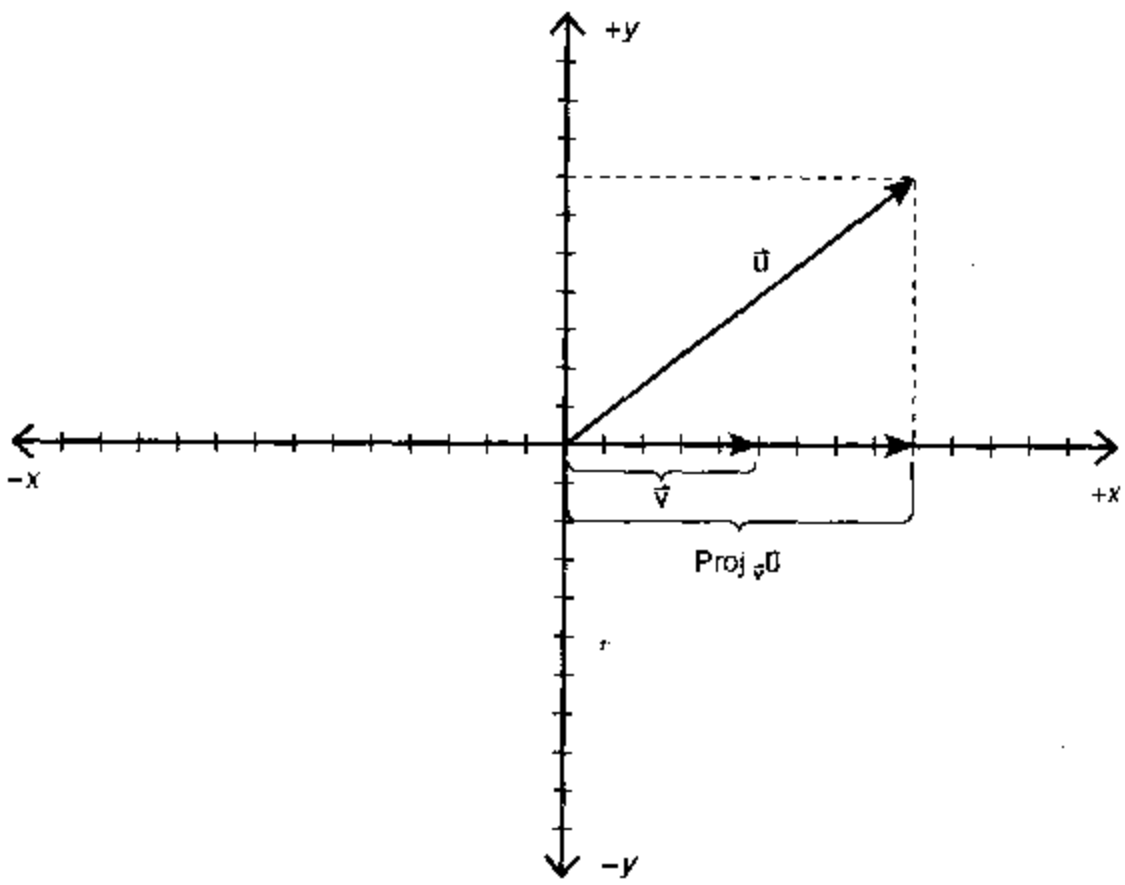


图 4.22　向量 u 在向量 v 上的投影

如图 4.22 所示，我们有向量 u 和 v，需要计算向量 u 在 v 上的投影的长度。如果需要向量，可以用投影长度乘以方向与向量 v 相同的单位向量。要计算投影长度，可使用点积 $u.v$，但 v 的长度必须为 1.0；也就是说，向量 v 必须被归一化。因此，计算 u 在 v 上投影向量的公式如下：

$$\text{Proj}_v\ u = \frac{(u \cdot v) * v}{|v| * |v|}$$

下面解释一下这个公式。分子中 $u.v$ 的含义如下：

$$u \cdot v = |u| * |v| * \cos(\theta)$$

也就是说，$u.v$ 等于向量 u 的长度乘以向量 v 的长度，再乘以它们之间夹角的余弦。在这里，我们需要计算 u 在 v 上的投影长度，但必须将向量 v 转换为单位向量——这正是分母中|v|之一的作用。因此，公式中计算投影长度的部分 $u.v/|v|$可简化为：

$$u \cdot v = |u| * 1 * \cos(\theta) = |u| * \cos(\theta)$$

它表明，当角度θ为 0（即向量 u 与 v 平行）时，$|u|*\cos(\theta)$变成$|u|$，这是正确的。现在，还有最后一个问题需要解决：我们需要的是投影向量，而不是投影向量的长度。这很容易，只需将投影向量的长度乘以方向与 v 相同的单位向量即可，结果如下：

$$\text{Proj}_v\ u = \frac{(u \cdot v) * v}{|v| * |v|}$$

这里旨在演示如何用点积解决投影问题。基本上，要计算一个向量在另一个向量上的投影，都可以使用点积；但必须将测试向量（这里为 v）转换为单位向量。

点积满足的乘法定律

标量乘法满足结合律、分配律和交换律；向量乘法可能并不满足这些定律，下面列出点积满足的乘法定律：

给定向量 u、v 和 w 以及标量 k：

```
(a)u.v = v.u
(b)u.(v + w)=(u.v + u.w)
(c)k*(u.v)=(k*u).v = (u.(k*v))
```

注意：点积的优先级高于加法。

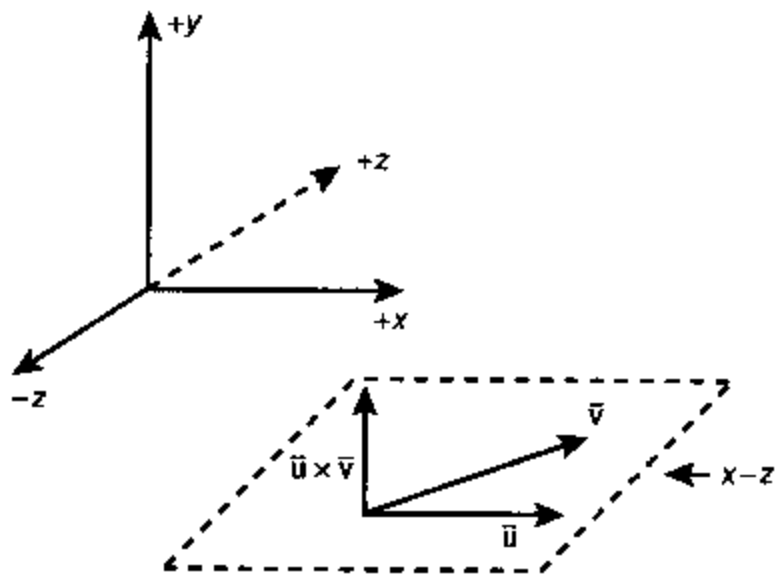
4.5.7　叉积

另一种向量乘法是叉积。然而，仅当向量包含 3 个或更多分量时，叉积才有意义。因此，这里将以 3D 向量为例进行讨论。给定 $u = <u_x, u_y, u_z>$和 $v = <v_x, v_y, v_z>$，叉积 $u \times v$ 的定义如下：

$$u \times v = |u| * |v| * \sin(\theta) * n$$

下面逐项分析这个公式。$|u|$为向量 u 的长度，$|v|$为向量 v 的长度，$\sin(\theta)$是两个向量之间夹角的正弦。因此，$|u|*|v|*\sin(\theta)$是一个标量，即是一个数值。然后，我们将它与 n 相乘，但 n 是什么呢？n 是一个单位法线向量，即它与向量 u 和 v 都垂直，且长度为 1.0。图 4.23 图示了这种乘法。

因此，根据叉积可以知道向量 u 和 v 之间的夹角以



$\bar{u} \times \bar{v}$ 与 $\bar{u}$ 和 $\bar{v}$ 都垂直，如果 $\bar{u}$ 和 $\bar{v}$ 都在 $x-z$ 平面内，则 $\bar{u} \times \bar{v}$ 与 y 轴平行

图 4.23　叉积

及 u 和 v 的法线向量。然而，如果没有另一个公式，将无法得到任何信息。问题是，如何计算 u 和 v 的法线向量呢？答案是使用叉积的另一种定义。

叉积还定义为一种非常特殊的向量积。然而，如果不使用矩阵，将难以描述这种定义。要计算 u 和 v 的叉积（$u \times v$），可以建立一个这样的矩阵：

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}$$

其中 i、j、k 分别是与 x 轴、y 轴、z 轴平行的单位向量。

要计算 u 和 v 的叉积，执行下面的乘法：

$$\mathbf{n} = (u_y * v_z - v_y * u_z) * \mathbf{i} - (u_x * v_z - v_x * u_z) * \mathbf{j} + (u_x * v_y - v_x * u_y) * \mathbf{k}$$

n 是三个标量分别乘以三个相互垂直（即分别与 x 轴、y 轴和 z 轴平行）的单位向量的线性组合。因此，可以省略 i、j、k，将上述公式表示为：

$$\mathbf{n} = \langle u_y * u_z - v_y * u_z,\ -u_x * v_z + v_x * u_z,\ u_x * v_y - v_x * u_y \rangle$$

n 是向量 u 和 v 的法线向量，但不一定是单位向量（如果 u 和 v 都是单位向量，n 也将是单位向量），因此必须归一化以得到 n。完成这一步后，就可以其代入到前面的叉积方程中，执行所需的计算。

然而，在实际应用中，很少有人使用公式 $u \times v = |u| * |v| * \sin(\theta) * n$，而只是使用矩阵形式来计算法线向量，因为θ通常是未知的。这里再次表明了对向量进行归一化在 3D 图形学中的重要性，您将使用归一化向量来进行光照计算、定义平面、比较多边形朝向、进行碰撞检测等。

叉积的乘法定律

下面列出叉积满足的乘法定律。

给定向量 u、v 和 w 以及标量 k：

(a) $u \times v = -(v \times u)$（非常重要）
(b) $u \times (v + w) = (u \times v) + (u \times w)$
(c) $(u + v) \times w = (u \times w) + (v \times w)$
(d) $k * (u \times v) = (k * u) \times v = u \times (k * v)$

4.5.8 零向量

虽然您不会经常使用零向量，但它的确存在。零向量的长度为零，没有方向，仅仅是一个点。因此，2D 零向量为<0，0>，3D 零向量为<0，0，0>，在维数更多的空间中，零向量与此类似。

4.5.9 位置和位移向量

接下来介绍位置和位移向量，跟踪几何实体，如直线、线段、曲线等时，它们很有用。图 4.24 所示描述了一个可用于表示线段的位置向量。该线段从 **p1** 到 **p2**，$\mathbf{v_d}$ 是从 **p1** 到 **p2** 的位移向量，v 是方向与从 **p1** 到 **p2** 向量相同的单位向量。可以创建向量 **p** 来跟踪该线段。从数学上说，向量 **p** 如下：

$$\mathbf{p} = \mathbf{p1} + t * \mathbf{v}$$

其中 t 是一个取值范围为 0 到$|v_d|$的参数。如果 $t=0$，则：

$$\mathbf{p} = \mathbf{p1} + 0 * \mathbf{v} = \langle \mathbf{p1} \rangle = \langle p1_x,\ p1_y \rangle$$

因此 $t=0$ 时，**p** 指向线段的起点。另一方面，如果 $t=v_d$，则：

$$\begin{aligned} \mathbf{p} &= \mathbf{p1} + |\mathbf{v_d}| * \mathbf{v} = \mathbf{p1} + \mathbf{v_d} = \langle \mathbf{p1} + \mathbf{v_d} \rangle \\ &= \langle p1_x + v_x,\ p1_y + v_y \rangle \\ &= \mathbf{p2} = \langle p2_x,\ p2_y \rangle \end{aligned}$$

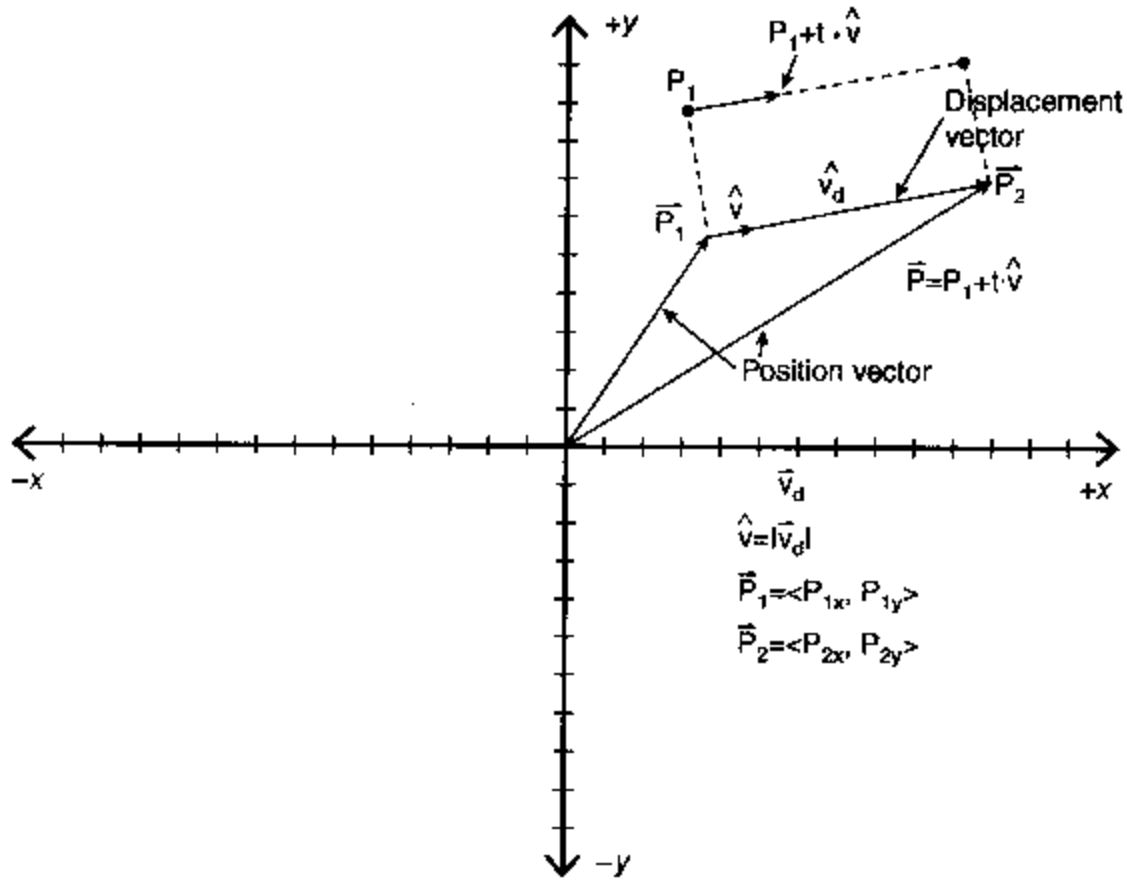


图 4.24　位置和位移向量

4.5.10　用线性组合表示的向量

正如您在向量叉积中见到的，向量还可以这样表示：

```
U = ux*i + uy*j + uz*k
```

其中 i、j、k 分别为与 x、y、z 轴平行的单位向量。这没什么特别的，它只是您可能需要知道的另一种向量表示法。所有运算的工作原理仍然相同。例如：

假设 $u = 3i + 2j + 3k$，$v = -3i - 5j + 12k$，则：

```
u + v = 3i + 2j + 3k - 3i - 5j + 12k
      = 0i - 3j + 15k = <0, -3, 15>
```

4.6　矩阵和线性代数

3D 图形学涉及对多组数值进行数学运算，使用矩阵可简化数据的表示和变换处理。

矩阵是一个矩形阵列，有指定行和列。通常说矩阵为 $m \times n$，表示它有 m 行和 n 列。$m \times n$ 也叫矩阵的维数。例如，下面的矩阵 A 为 2×2 维：

```
A = |1  4|
    |9 -1|
```

本书使用粗体大写字母来表示矩阵。通常，大多数人都使用粗体大写字母来表示矩阵，但也有人用斜体大写字母来表示矩阵。在上面的范例中，第 1 行为<1 4>，第二行位<9 –1>。下面是一个 3×2 的矩阵：

```
    |5    6|
B = |2    3|
    |100 -7|
```

下面是一个 2×3 的矩阵：

```
C = | 3 5  0|
    |-8 12 4|
```

要定位矩阵中的第<i，j>个元素，只需找到第 i 行第 j 列的值即可。然而，在大多数数学书中，对矩阵元素从 1 开始计数，而不像在计算机程序中那样从 0 开始计数。本书从 0 开始计数，这样使用 C/C++矩阵时将更自然。例如，在下面的 3×3 矩阵中，元素名中包含下标：

$$A=\begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

有关矩阵本身及元素的标记规则就这些。您可能会问，“矩阵从哪里来？”矩阵是用于表示方程组的数学工具。如果有一个下面这样的方程组：

```
3*x + 2*y = 1
4*x - 9*y = 9
```

书写变量很麻烦。既然知道方程是关于 x 和 y 的，为什么还要书写这些变量呢？为什么不使用一种更加紧凑的各种，其中只包含方程系数呢？这就是矩阵的作用。对于上面的范例，可以使用三个矩阵来表示三套不同的值，然后同时或分别使用它们。

1. 系数矩阵：

```
3*x + 2*y = 1
4*x - 9*y = 9
A = |3  2|
    |4 -9|
```

2. 变量矩阵：

```
3*x + 2*y = 1
4*x - 9*y = 9
X = |x|
    |y|
```

3. 常量矩阵：

```
3*x + 2*y = 1
4*x - 9*y = 9

B = |1|
    |9|
```

有了上面这些矩阵后，便可以将重点放在系数矩阵 A 上，而不考虑其他内容。另外，还可以使用矩阵格式书写方程组：

```
A*X = B
```

执行数学运算（乘法）后，结果如下：

```
3*x + 2*y = 1
4*x - 9*y = 9
```

然而，如何执行数学运算呢？我们知道，向量乘法并不直观，那么矩阵运算呢？首先来看一下在矩阵语境中“1”的含义。

4.6.1 单位矩阵

在任何一个数学系统中，首先需要定义的是 1 和 0。在矩阵数学中，也有类似于这样的值，与 1 对应的是单位矩阵（identity matrix）I，其定义是：主对角线上所有元素都为 1，其他元素为 0。单位矩阵具有如下性质：

```
A*I = I*A = A
```

注意：矩阵 I 和 A 必须具有相同的维数。

由于矩阵可以为任意维数，显然有无数的单位矩阵，但有一个限制：所有单位矩阵都必须是方形的，即维数为 $m \times m$，其中 $m >= 1$。下面分别是 2×2 和 3×3 的单位矩阵：

$$\mathbf{I}_2 = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$$

$$\mathbf{I}_3 = \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

从技术上说，单位矩阵并不相当于 1，但在矩阵乘法中相当于 1。

第二种基本矩阵是零矩阵（zero matrix）**Z**，在加法运算和乘法运算中，它相当于 0。它是一个 $m \times n$ 矩阵，所有元素都为 0。

$$\mathbf{Z}_{3\times3} = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

$$\mathbf{Z}_{1\times2} = \begin{vmatrix} 0 & 0 \end{vmatrix}$$

关于零矩阵惟一有意义的事情是，在矩阵加法和乘法中，它有标量 0 的属性。除此之外，它没有其他用途。

4.6.2　矩阵加法

矩阵加法和减法将两个矩阵的每个元素相加或相减，得到结果矩阵中对应的元素。关于矩阵加法和减法的惟一规则是，两个矩阵必须有相同的维数。下面是两个范例：

假设：

$$\mathbf{A} = \begin{vmatrix} 1 & 5 \\ -2 & 0 \end{vmatrix} \qquad \mathbf{B} = \begin{vmatrix} 13 & 7 \\ 5 & -10 \end{vmatrix}$$

则：

$$\mathbf{A} + \mathbf{B} = \begin{vmatrix} 1 & 5 \\ -2 & 0 \end{vmatrix} + \begin{vmatrix} 13 & 7 \\ 5 & -10 \end{vmatrix} = \begin{vmatrix} (1+13) & (5+7) \\ (-2+5) & (0-10) \end{vmatrix} = \begin{vmatrix} 14 & 12 \\ 3 & -10 \end{vmatrix}$$

$$\mathbf{A} - \mathbf{B} = \begin{vmatrix} 1 & 5 \\ -2 & 0 \end{vmatrix} + \begin{vmatrix} 13 & 7 \\ 5 & -10 \end{vmatrix} = \begin{vmatrix} (1-13) & (5-7) \\ (-2-5) & (0-(-10)) \end{vmatrix} = \begin{vmatrix} -12 & -2 \\ -7 & 10 \end{vmatrix}$$

4.6.3　矩阵的转置

有时候，将矩阵的行转为列，可方便地表示另一个矩阵。例如，对于 1×3 矩阵 $M = [x\ y\ z]$，其转置矩阵 $\mathbf{M}^t$ 是一个 3×1 矩阵：

$$\mathbf{M}^t = \begin{vmatrix} x \\ y \\ z \end{vmatrix}$$

对于 $m \times n$ 矩阵，通过交换行和列可以得到其转置矩阵。例如，对于如下所示的 3×2 矩阵 **A**：

$$\mathbf{A} = \begin{vmatrix} 5 & 6 \\ 2 & 3 \\ 100 & -7 \end{vmatrix}$$

其转置矩阵为 2×3 矩阵：

$$A^t = \begin{vmatrix} 5 & 2 & 100 \\ 6 & 3 & -7 \end{vmatrix}$$

有关转置矩阵就介绍到这里。计算逆矩阵和矩阵乘法时，有时候需要用到转置矩阵。

4.6.4 矩阵乘法

矩阵乘法有两种：标量-矩阵乘法；矩阵-矩阵乘法。标量-矩阵乘法是用标量来乘以矩阵，只需将矩阵的每个元素乘以该标量即可，矩阵可以为任何维数。下面是 3×3 矩阵与标量相乘的情况：

假设 k 是一个任意实数。

如果：

$$\text{Let } \mathbf{A} = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

则：

$$\text{Then } k*\mathbf{A} = k* \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix} = \begin{vmatrix} k*a_{00} & k*a_{01} & k*a_{02} \\ k*a_{10} & k*a_{11} & k*a_{12} \\ k*a_{20} & k*a_{21} & k*a_{22} \end{vmatrix}$$

下面是一个范例：

$$3* \begin{vmatrix} 1 & 4 \\ -2 & 6 \end{vmatrix} = \begin{vmatrix} (3*1) & (3*4) \\ (3*(-2)) & (3*6) \end{vmatrix} = \begin{vmatrix} 3 & 12 \\ -6 & 18 \end{vmatrix}$$

数学知识：将矩阵方程两边同时乘以一个标量后，该方程仍然成立。因为在方程组两边同时乘以一个常量系数后，方程仍然成立。

第二种形式是真正的矩阵乘法，其数学原理要稍微复杂些，但可以将一个矩阵视为对另一个矩阵执行运算的“运算符”。

要将矩阵 **A** 和 **B** 相乘，它们的内维数（inner dimension）必须相等，换句话说，如果 **A** 为 $m \times n$ 矩阵，则 **B** 必须为 $n \times r$ 矩阵。

其中 m 和 r 可以相等，也可以不相等。例如，可以执行下列矩阵的乘法：2×2 乘以 2×2、3×2 乘以 2×3、4×4 乘以 4×5，但不能将 3×3 矩阵乘以 2×4 矩阵，因为它们内维数 3 和 2 不相等。矩阵相乘得到的结果矩阵的大小为两个矩阵的外维数。例如，将 2×3 矩阵乘以 3×4 矩阵时，将得到一个 2×4 矩阵。

矩阵乘法难以用语言来描述，请参考图 4.25，其中给出了矩阵乘法的技术描述。

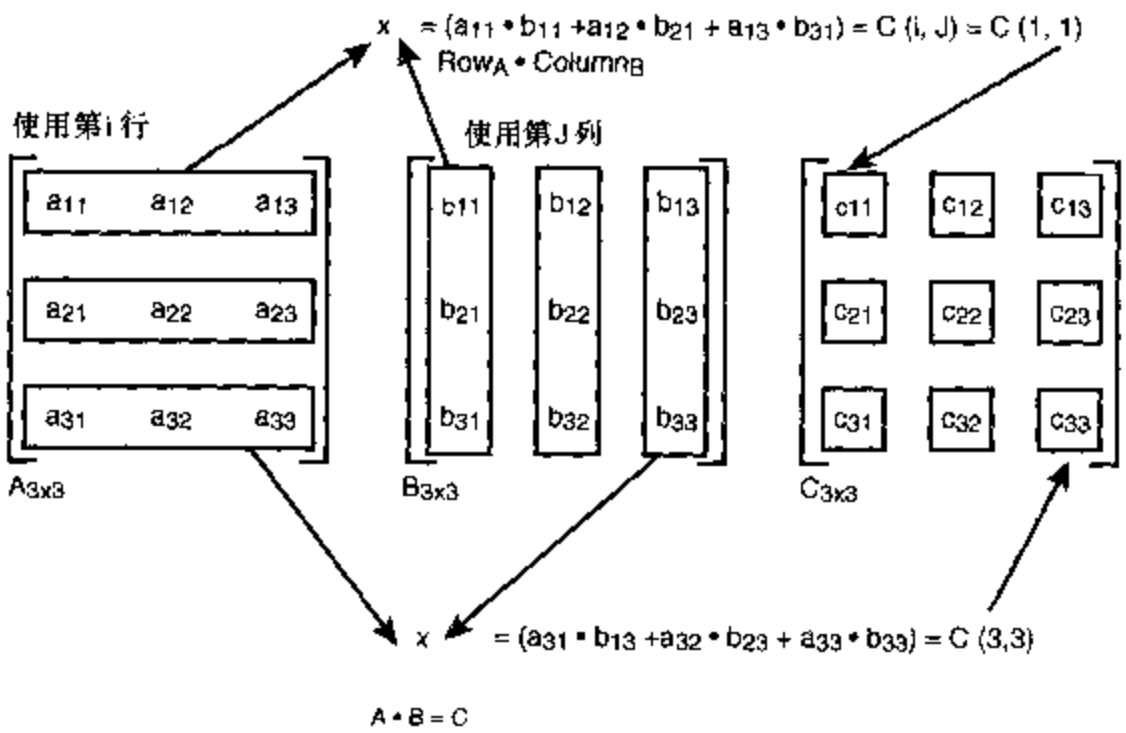


图 4.25 矩阵乘法的工作原理

给定矩阵 **A** 和 **B**，要将它们相乘得到结果矩阵 **C**（**A**×**B**=**C**），需要用矩阵 **A** 的一行乘以矩阵 **B** 中的一列，然后将每对元素的积相加（即点积）。下面是一个 2×2 矩阵乘以 2×3 矩阵的例子。

假设

$$\mathbf{A} = \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} \quad \mathbf{B} = \begin{vmatrix} 1 & 3 & 5 \\ 6 & 0 & 4 \end{vmatrix}$$

$$\mathbf{C} = \mathbf{A} \times \mathbf{B} = \begin{vmatrix} (1*1+2*6) & (1*3+2*0) & (1*5+2*4) \\ (3*1+4*6) & (3*3+4*0) & (3*5+4*4) \end{vmatrix}$$

$$= \begin{vmatrix} 13 & 3 & 13 \\ 27 & 9 & 31 \end{vmatrix}$$

提示：请注意其中以粗体显示的（1*1 + 2*6）。它和矩阵 **C** 的其他所有元素，都是矩阵 **A** 中一个行向量与矩阵 **B** 中一个列向量的点积。因此，$\mathbf{C}_{ij}$ 是矩阵 **A** 中第 i 行与矩阵 **B** 中第 j 列的点积。

4.6.5　矩阵运算满足的定律

向量满足一些与加法和乘法相关的结合律、交换律、分配律等，矩阵也满足某些定律，它们是：

（a）**A** + **B** = **B** + **A**（加法交换律）

（b）**A** + (**B** + **C**) = (**A** + **B**) + **C**（加法结合律）

（c）**A***(**B***C**) = (**A***B**)*C**（乘法结合律）

（d）**A***(**B** + **C**) = **A***B** + **A***C**（分配律）

（e）k*(**A** + **B**) = k*A + k*B（分配律）

（f）(**A** + **B**)*C = A*C + B*C（分配律）

（g）**A***I = I*A = **A**（单位矩阵的乘法属性）

注意：上面所有的“+”都可以替换为“-”。

然而，下面的等式通常不成立：

(A*B)≠(B*A)

也就是说，矩阵乘法不满足交换律，因此在矩阵乘法中，矩阵的顺序相当重要。

注意：一种(A*B) = (B*A)的情况是，A 或 B 为单位矩阵。

4.7　逆矩阵和方程组求解

现在您知道如何创建和操作矩阵，但还需要学习计算另一个数学对象：逆矩阵。换句话说，对于标量 x，如果存在另一个标量 x^{-1}，使得 $x*x^{-1}=1$，则 x^{-1} 为 x 的倒数。计算标量的倒数很容易，给定不为 0 的实数 x，x^{-1} 的计算公式如下：

$$x^{-1} = 1/x$$

然而，计算逆矩阵复杂得多。例如，对于下面的矩阵方程，我们需要求解 **X**：

A*X = B

如果已知矩阵 **A** 的逆矩阵（$\mathbf{A}^{-1}$），可以用它来乘以方程两边：

$$(\mathbf{A}^{-1}*\mathbf{A})*\mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$$

我们知道，任何矩阵乘以其逆矩阵的结果为单位矩阵，因此：

$$(\mathbf{I})*\mathbf{X} = \mathbf{A}^{-1}*\mathbf{B}$$

而任何矩阵乘以单位矩阵的结果仍为该矩阵，因此 **X** 的解如下：

$$X = A^{-1}*B$$

在很多 3D 游戏编程领域，如图形学、物理学和动画中，以这种方式求解方程组非常重要。这里没有时间详细讨论如何计算通用矩阵 **A** 的逆矩阵，只介绍 2×2 矩阵的情况。

给定矩阵：

$$A = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

如果该矩阵的行列式（即 Det（**A**））不等于零，则其逆矩阵存在，且对应的方程组有惟一的解。矩阵 **A**（或任何 2×2 矩阵）的行列式的公式计算如下：

$$Det(A) = (a*d - b*c)$$

如果结果不为零，矩阵 **A** 的逆矩阵 A^{-1} 存在，其计算公式如下：

$$A^{-1} = \frac{1}{det(A)} * \begin{vmatrix} d & -b \\ -c & a \end{vmatrix}$$

将行列式代入其中，结果如下：

$$A^{-1} = \begin{vmatrix} d/(a*d - b*c) & -b/(a*d - b*c) \\ -c/(a*d - b*c) & a/(a*d - b*c) \end{vmatrix}$$

下面用一个方程组来验证。假设有如下两个线性方程：

$$\begin{aligned} 3*x+5*y &= 6 \\ -2*x+2*y &= 4 \end{aligned}$$

要使用矩阵技术来求解 x 和 y，首先提取系数矩阵 **A**、变量矩阵 **X** 和常量矩阵 **B**：

$$A = \begin{vmatrix} 3 & 5 \\ -2 & 2 \end{vmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

$$X = \begin{vmatrix} x \\ y \end{vmatrix}$$

$$B = \begin{vmatrix} 6 \\ 4 \end{vmatrix}$$

因此，需要在下列方程中求解 **X**：

$$A*X = B,\ X = A^{-1}*B$$

计算矩阵 **A** 的逆矩阵，如下所示：

$$A = \begin{vmatrix} 3 & 5 \\ -2 & 2 \end{vmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

$$(a*d - b*c) = (3*2-(-2)*5) = (6+10) = 16$$

$$A^{-1} = \begin{vmatrix} d/(a*d-b*c) & -b/(a*d-b*c) \\ -c/(a*d-b*c) & a/(a*d-b*c) \end{vmatrix}$$

$$A^{-1} = \begin{vmatrix} 2/16 & -5/16 \\ 2/16 & 3/16 \end{vmatrix}$$

将它代入矩阵方程中：

$$X = A^{-1}*B$$

$$X = \begin{vmatrix} 2/16 & -5/16 \\ 2/16 & 3/16 \end{vmatrix} * \begin{vmatrix} 6 \\ 4 \end{vmatrix} = \begin{vmatrix} (2/16)*6+(4)*(-5)/16 \\ (2/16)*6+(4)*3/16 \end{vmatrix}$$

$$= \begin{vmatrix} 12/16 & -20/16 \\ 12/16 & +12/16 \end{vmatrix}$$

$$= \begin{vmatrix} -8/16 \\ 24/16 \end{vmatrix} = \begin{vmatrix} -0.5 \\ 1.5 \end{vmatrix}$$

因此，$x = -0.5$，而 $y = 1.5$。将它们的值代入到第一个方程中，结果如下：

```
  3*x   +   5*y      = 6
3*(-0.5)+   5*(1.5)  = 6
 -1.5   +   7.5      = 6
```

这表明 x 和 y 的解是正确。为进一步确认，将它们代入第二个方程：

```
-2*x       + 2*y      = 4
-2*(-0.5)  + 2*(1.5)  = 4
   1.0     +      3   = 4
```

完全正确！

4.7.1　克来姆法则

使用克来姆法则求解包含三个或更多变量的大型方程组的问题在于，计算系数矩阵的逆矩阵涉及很大的计算量。它可能相当复杂，通常需要使用诸如高斯消元法等技术来重复处理方程组。

注意：高斯消元法对系数矩阵进行处理，将其转换为从中可以直接读取方程组解的形式。这种处理基于如下事实：可以对矩阵的任何行乘以一个系数，或将某一行的倍数加入到其他行。使用这些基本定律，可以逐步地将系数矩阵转换为这样：其中某一行只有一个非零系数，从而可以求解一个变量。然后继续该过程，或将已求解变量代入，来求解其他变量。

然而，对于三变量方程组，有一种基于克来姆法则的封闭式（closed form）解决方案，您可能听说过它。三变量方程组的克来姆法则如下：

给定方程组：

$$\begin{aligned} a_{00}*x + a_{01}*y + a_{02}*z &= b_0 \\ a_{10}*x + a_{11}*y + a_{12}*z &= b_1 \\ a_{20}*x + a_{21}*y + a_{22}*z &= b_2 \end{aligned}$$

可以将其转换为如下矩阵形式：

$$A = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

$$X = \begin{vmatrix} x \\ y \\ z \end{vmatrix}$$

$$B = \begin{vmatrix} b_0 \\ b_1 \\ b_2 \end{vmatrix}$$

现在需要计算矩阵 A 的行列式 Det（A），对于下列 3×3 矩阵：

$$A = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix}$$

其行列式计算公式如下：

$$Det(A) = a_{00}*a_{11}*a_{22} + a_{01}*a_{12}*a_{20} + a_{02}*a_{10}*a_{21} - a_{02}*a_{11}*a_{20} - a_{01}*a_{10}*a_{22} - a_{00}*a_{12}*a_{21}$$

这包含 12 次乘法和 5 次加法。

还有一种基于余因子扩展（cofactor expansion）计算矩阵行列式的方法，它相当复杂，这里只给出结果：

$$Det(A) = a_{00}*(a_{11}*a_{22} - a_{21}*a_{12}) - a_{01}*(a_{10}*a_{22} - a_{20}*a_{12}) + a_{02}*(a_{10}*a_{21} - a_{20}*a_{11})$$

这包含 9 次乘法和 5 次加法。

有了 3×3 矩阵的 Det（A）后，克来姆法则可以表示为一组求解 x、y、z 的商。对于每个变量（x、y、z），都有一个用于求解它的商，其中分子分别是用列向量 **B** 替换矩阵 **A** 的第一列、第二列和第三列得到的矩阵的行列式，分母都为 Det（A）。具体情况如下：

$$x = \frac{Det\begin{vmatrix} b_0 & a_{01} & a_{02} \\ b_1 & a_{11} & a_{12} \\ b_2 & a_{21} & a_{22} \end{vmatrix}}{Det(A)}$$

$$y = \frac{Det\begin{vmatrix} a_{00} & b_0 & a_{02} \\ a_{10} & b_1 & a_{12} \\ a_{20} & b_2 & a_{22} \end{vmatrix}}{Det(A)}$$

$$z = \frac{Det\begin{vmatrix} a_{00} & a_{01} & b_0 \\ a_{10} & a_{11} & b_1 \\ a_{20} & a_{21} & b_2 \end{vmatrix}}{Det(A)}$$

本书后面将编写软件来计算矩阵的行列式和求解包含两个和三个变量的方程组。

4.7.2 使用矩阵进行变换

学习有关矩阵的知识旨在使用它们执行 2D/3D 变换。我们将把要变换的点 **p** 乘以所需的变换矩阵。数学表示为：

$$\mathbf{p}' = \mathbf{p}*\mathbf{M}$$

其中 **p**'是变换后的点，**p** 是原始点，**M** 是变换矩阵。另外，还可以将多个变换矩阵相乘，来执行多次变换。也就是说，可以先将多个矩阵相乘，如下所示：

$$\mathbf{M} = \mathbf{M}_1*\mathbf{M}_2*\mathbf{M}_3*...\mathbf{M}_n$$

下面的变换是等价的：

$$\mathbf{p}' = \mathbf{p}*\mathbf{M} = \mathbf{p}*\mathbf{M}_1*\mathbf{M}_2*\mathbf{M}_3*...\mathbf{M}_n$$

不管将 **p** 乘以 **M** 还是 $\mathbf{M}_1*\mathbf{M}_2*\mathbf{M}_3*...\mathbf{M}_n$，结果都是相同的。例如，假设有三个变换 $\mathbf{M}_1$、$\mathbf{M}_2$ 和 $\mathbf{M}_3$，需要将它们应用于点 **p**，以得到 **p**'；这些变换可以是平移、旋转或投影。可以以每次一个的方式执行这一系列变换：

$p1 = p*M_1$

$p2 = p1*M_2$

$p' = p2*M_3$

注意，p1 和 p2 只是临时变量。

进行一系列变换是完全可以接受的。然而，如果有数千甚至数百万个点需要变换，这些中间步骤将浪费时间！可以通过替换，将这些变换组合起来，如下所示：

$p1 = p*M_1$

$p2 = p*M_1*M_2$

$p' = p*(M_1*M_2*M_3)$

因此，对 **p** 的变换可以表示为 **p***(**M**)，其中 **M** = (**M**$_1$*__M__$_2$*__M__$_3$)。可以一次性计算 **M** = (**M**$_1$*__M__$_2$*__M__$_3$)，然后使用它来进行变换，这样对于每个点都可以节省两次矩阵乘法！

当然，必须计算 **M**，但只需要计算一次，然后可以对多个点执行这种变换。将矩阵组合起来在 3D 流水线中非常有用，尤其是需要将物体空间中的 3D 顶点变换为屏幕空间中的点时。

注意：由于矩阵乘法通常不满足交换律，因此顺序非常重要，即 **M**$_1$*__M__$_2$*__M__$_3$ ≠ **M**$_3$*__M__$_2$*__M__$_1$。

4.7.3　齐次坐标

接下来要介绍的主题是齐次坐标。在大多数情况下，我们将用顶点列表来表示 3D 物体，每个顶点有三个分量：**p** (*x*, *y*, *z*)，或表示为向量<*x*, *y*, *z*>或表示为行矩阵[*x y z*]，它们是等价的，只是表示方法不同而已。将 3D 点与矩阵相乘时，我将使用最后一种表示法。为充分利用矩阵变换的功能，我们将使用齐次坐标来表示 2D 和 3D 空间中的点，以便能够使用矩阵来执行各种变换。要将 2D 或 3D 点转换为齐次坐标，需要加上 *w* 分量，如下所示：

3D 点[*x y z*]的齐次坐标为：

[x y z w]

2D 点[x y]的齐次坐标为：

[x y w]

其中 *w* 用于齐次化坐标，但它不是虚假的。要将齐次坐标转换为常规坐标，必须除以 *w*。例如，3D 空间的情况如下：

给定齐次坐标：

[x y z w]

其常规坐标为：

[x/w y/w z/w]

如果 *w* = 1，这种变换将不会改变 *x*、*y*、*z*。那么增加 *w* 有何用途呢？*w* 让我们能够表示无穷大的概念。例如，当 *w* 趋近于 0 时，*x*/*w* 将趋近于∞。换句话说，任何分数的分母趋近于 0 时，商将趋近于无穷大。另外，齐次坐标来源于齐次方程。例如，请看如下线性方程：

a*x + b*y + c = 0

这是一个一次线性方程，其中两个变量项的次数都为 1，但常量项的次数为 0。如果希望所有项的次数都相同（齐次化），该如何办呢？可以用 *x*/*w* 代替 *x*，用 *y*/*w* 代替 *y*，将这个多项式转换为齐次多项式：

a*x/w + b*y/w + c = 0

为消除分母中的 *w*，将两边乘以 *w*，结果如下：

```
a*x + b*y + c*w = 0
```

现在每一项都是一次的，因此多项式是齐次的。如果使用齐次坐标表示几何对象，可确保能够执行所有的变换，包括投影变换和平移。这里不再赘述，但需要指出的是，在很多情况下，我们将使用哑变量 $w = 1$，以便能够使用矩阵表示所有的变换。实际上，我们可能不会在数据结构中存储 w，因为大多数情况下，都将 $w = 1$；当 $w = 1$ 时，齐次坐标[$x\ y\ z\ w$]将就是[$x\ y\ z$]。这里只想让您知道这种数学技巧的来源。

下一章将更详细地全面介绍 3D 变换，这里只是简要地描述一些最基本的 3D 变换（平移、缩放和旋转）以及为支持这三种变换为何需要使用齐次坐标。

4.7.4 应用矩阵变换

下面演示我们将对 2D 和 3D 数据执行的一些基本矩阵变换。这些实例都是 3D 的，但在大多数情况下，只需丢掉 z 分量，就可以推导出 2D 版本。注意，在所有的范例中，w 都为 1。

1. 平移

要在 3D 空间中执行平移，需要将点 $\mathbf{p}(x, y, z)$ 平移到新的位置 $\mathbf{p}'(x+dx, y+dy, z+dz)$。图 4.26 说明了 x-y 平面中的简化情况。下面的 4×4 矩阵用于执行平移变换：

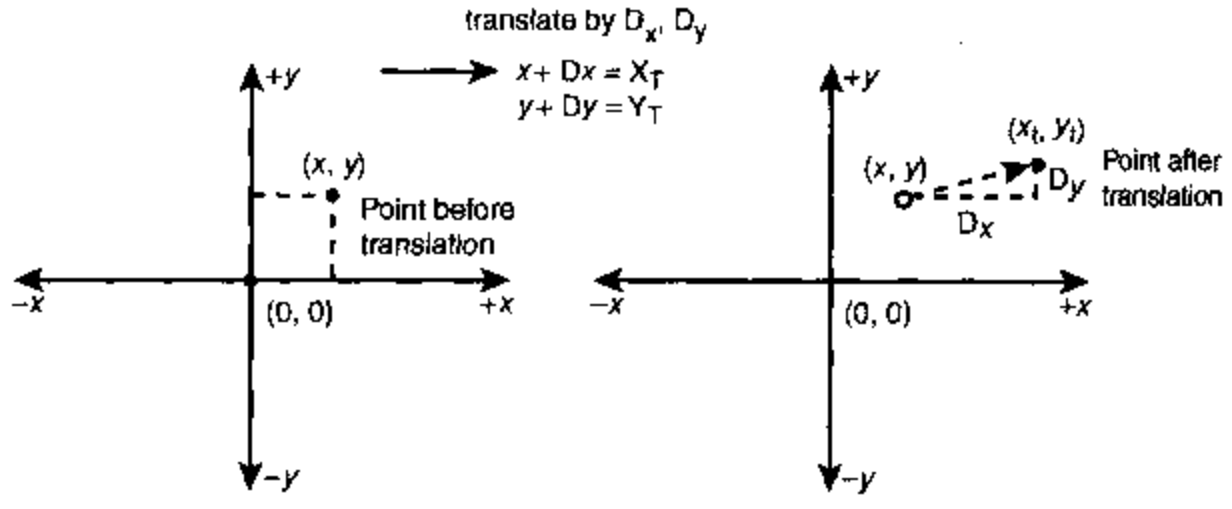


图 4.26 2D 空间平移的几何描述

$$\mathbf{M}_t = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{vmatrix}$$

给定 $\mathbf{p} = [x\ y\ z\ 1]$，执行如下平移变换：

$$\mathbf{p}' = \mathbf{p}*\mathbf{M}_t = [x\ \ y\ \ z\ \ 1]* \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{vmatrix}$$

```
= [(x+1*dx) (y+1*dy) (z+1*dz) (1*1)]

= [(x+dx) (y+dy) (z+dz) 1]
```

然后用 w 分量（1.0）除以其他三个分量，结果如下：

```
= [(x+dx)/1 (y+dy)/1 (z+dz)/1]

= p'((x+dx), (y+dy), (z+dz))
```

注意：实际上，编写数学库时，我们不想浪费空间，也不想每次都除以 1.0，因此将使用 4×3 矩阵进行大多数变换，并假设 $w = 1$。

如果取出前三个分量，结果如下：

```
x' = x+dx
y' = y+dy
z' = z+dz
```

这正是我们想要的结果。

2. 平移变换矩阵的逆矩阵

只需要将矩阵 $\mathbf{M}_t$ 中 dx、dy、dz 的符号反转，便可以得到 $\mathbf{M}_t$ 的逆矩阵：

$$\mathbf{M}_t^{-1} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -dx & -dy & -dz & 1 \end{vmatrix}$$

为验证这一点，下面来计算 $\mathbf{M}_t * \mathbf{M}_t^{-1}$，结果应该为 $\mathbf{I}_{4\times4}$：

$$\mathbf{M}_t * \mathbf{M}_t^{-1} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{vmatrix} * \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -dx & -dy & -dz & 1 \end{vmatrix}$$

$$= \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \mathbf{I}_{4\times4}.$$

注意：对于常用变换矩阵，以一种高效率的方式计算出其逆矩阵，在 3D 图形中是非常重要的，尤其是对于相机变换。

3. 缩放

要相对于原点缩放某个点，只需要将点 $\mathbf{p}(x, y, z)$ 的各个分量分别乘以 x 轴、y 轴、z 轴的缩放因子 sx、sy、sz。如图 4.27 说明了点位于 x-y 平面中的简化情况。在进行缩放操作时，不想进行平移。下面是用于缩放变换的矩阵：

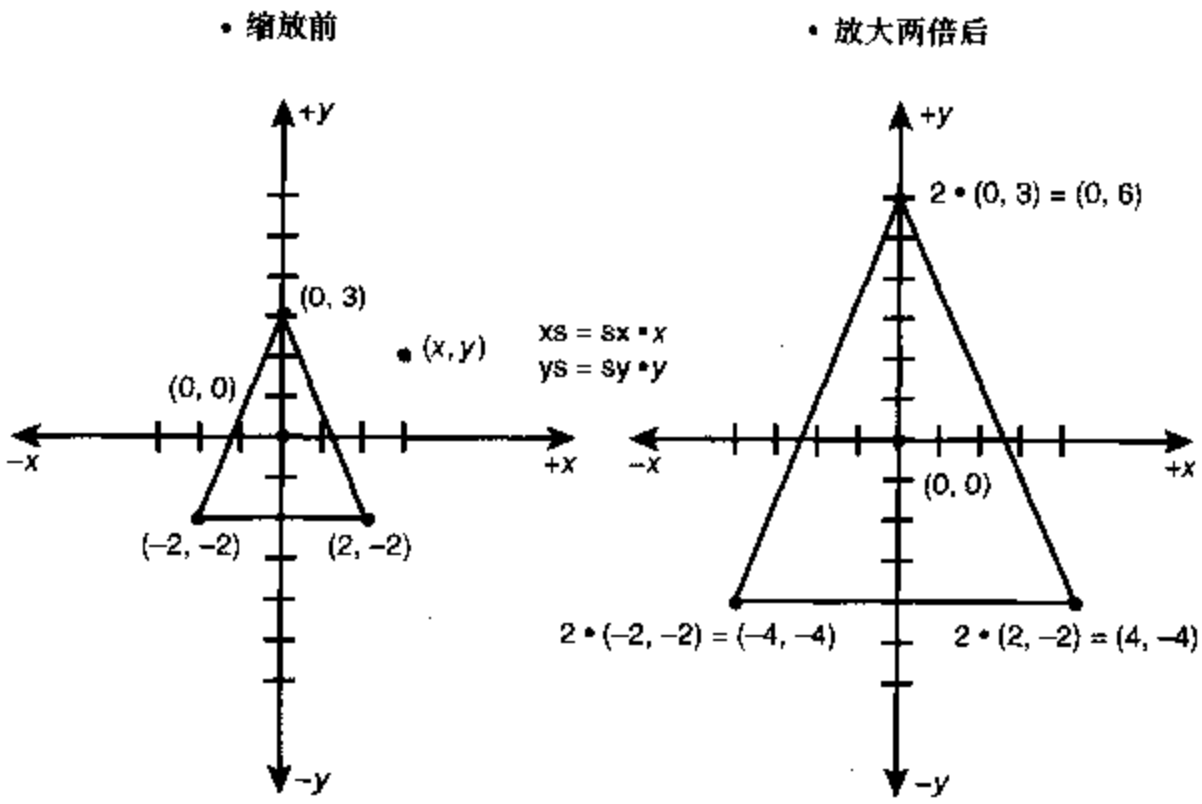


图 4.27　缩放操作

$$\mathbf{M}_s = \begin{vmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

给定 $\mathbf{p} = [x\ y\ z\ 1]$，执行如下变换：

$$\mathbf{p}' = \mathbf{p}*\mathbf{M}_s = [x\ \ y\ \ z\ \ 1]* \begin{vmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

```
=[(x*sx) (y*sy) (z*sz) (1*1)]

=[(x*sx) (y*sy) (z*sz) 1]
```

然后用分量 w 除以前三个分量，结果如下：

```
= [(x*sx)/1 (y*sy)/1 (z*sz)/1]

= p'((x*sx), (y*sy), (z*sz))
```

同样，这也正是我们想要的缩放结果；即：

```
x' = sx*x
y' = sy*y
z' = sz*z
```

数学知识：注意变换矩阵右下角的 1。从技术上说，它并不是必需的，因为就缩放而言，并不需要使用第 4 列的结果，因此将浪费一些 CPU 周期。同样，如平移变换范例中所述，使用第 4 个分量来实现 3D 空间点的齐次坐标时，必须使用 4x4 矩阵，以确保在数学上是正确的。后面我们将丢掉任何不需要的内容。

4. 缩放变换矩阵的逆矩阵

虽然不如平移变换矩阵的逆矩阵那样常用，但计算缩放变换矩阵的逆矩阵也是非常方便的。要计算缩放矩阵 $\mathbf{M}_s$ 的逆矩阵，只需将矩阵 $\mathbf{M}_s$ 中的缩放系数 sx、sy、sz 替换为各自的倒数即可：

$$\mathbf{M}_s^{-1} = \begin{vmatrix} 1/sx & 0 & 0 & 0 \\ 0 & 1/sy & 0 & 0 \\ 0 & 0 & 1/sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

为检查逆矩阵是否正确，下面来计算 $\mathbf{M}_s*\mathbf{M}_s^{-1}$，它应该为 $\mathbf{I}_{4\times4}$：

$\mathbf{M}_s*\mathbf{M}_s^{-1}=$

$$= \underbrace{\begin{vmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}}_{\mathbf{M}_s} * \underbrace{\begin{vmatrix} 1/sx & 0 & 0 & 0 \\ 0 & 1/sy & 0 & 0 \\ 0 & 0 & 1/sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}}_{\mathbf{M}_s^{-1}}$$

$$= \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \mathbf{I}_{4\times4}.$$

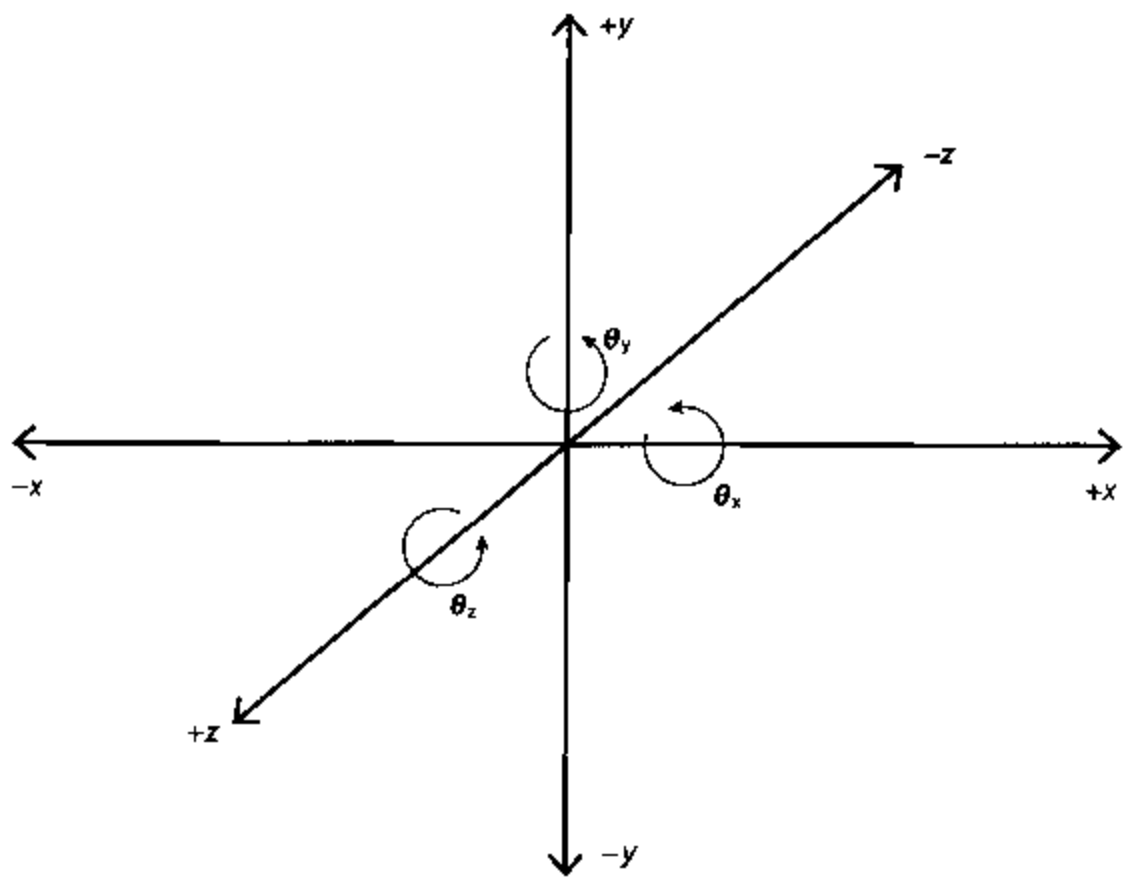
注意：使用高斯消元法或其他方法来计算逆矩阵非常复杂和繁琐，这里从几何属性的角度来考虑问题，进行计算逆矩阵。一般而言，变换为 2D 或 3D 空间中的几何变换时，计算其逆变换都非常简单，通常只需

将几何变换反转即可。请记住这一点，看看您能否想出计算旋转矩阵的逆矩阵的方法。

5. 旋转

旋转矩阵是所有变换矩阵中最复杂的，因为它充斥了三角函数。基本上，我们将使用旋转方程来对点进行旋转。为此，必须看一下旋转方程，提取其中的运算符，然后将它们放到矩阵中。另外，我们不想同时进行平移，因此最后一行的前三个元素总是为 0。

还有一个技术细节：在 3D 坐标系中，可以绕三个轴进行旋转：x 轴、y 轴、z 轴，如图 4.28 所示。我们将先考虑绕 z 轴旋转的情况，然后将其他作为模型来创建绕 x 轴和 y 轴旋转的变换矩阵。重要的是，绕任何轴的旋转时，该轴对应的分量保持不变。



右手坐标系

图 4.28　旋转轴

警告：θ为正时，对应的旋转方向取决于坐标系类型。如果使用的是右手坐标系，对应的旋转方向为逆时针；如果使用的是左手坐标系，对应的旋转方向为顺时针。

下面分别是绕 x 轴、y 轴、z 轴旋转对应的旋转方程：

绕 z 轴旋转：

$$M_z = \begin{vmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

注意：如果删除矩阵的最后一行和最后一列，将得到在平面中的 2D 旋转矩阵。

绕 x 轴旋转：

$$M_x = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

绕 y 轴旋转：

$$\mathbf{M}_y = \begin{vmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

下面来看一个绕 z 轴旋转的范例，以检查变换矩阵是否正确。

给定点 $\mathbf{p}=[x\ y\ z\ 1]$，绕 z 轴旋转角度 θ：

$$\mathbf{p}' = \mathbf{p}*\mathbf{M}_z = [x\ \ y\ \ z\ \ 1]* \begin{vmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$p'=[(x*\cos\theta - y*\sin\theta)\ (x*\sin\theta + y*\cos\theta)\ z\ (1*1)]$$

将前三个分量分别除以分量 w，结果如下：

$$\mathbf{p}'=((x*\cos\theta-y*\sin\theta),\ (x*\sin\theta + y*\cos\theta),\ z)$$

注意，z 将保持不变，与期望的相同。

6. 旋转矩阵的逆矩阵

$\mathbf{M}_x$、$\mathbf{M}_y$ 和 $\mathbf{M}_z$ 的逆矩阵的计算方法相同，因此这里只介绍如何计算 $\mathbf{M}_z$ 的逆矩阵。要计算 $\mathbf{M}_z$ 的逆矩阵，可以使用两种方法：一种方法基于几何学，另一种方法基于线性代数。首先使用几何方法来计算逆矩阵。将物体绕 z 轴旋转角度 θ 后，要将它恢复到原来的位置，只需将它旋转角度 $-\theta$ 即可。因此要计算旋转矩阵的逆矩阵，只需将旋转矩阵中的 θ 替换为 $-\theta$。因此，逆矩阵 $\mathbf{M}_z^{-1}$ 如下：

$$\mathbf{M}_z^{-1} = \begin{vmatrix} \cos-\theta & \sin-\theta & 0 & 0 \\ -\sin-\theta & \cos-\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

下面进一步简化，以消除 $-\theta$。根据 4.4.3 一节中列出的反射恒等式可知：

$$\sin(-\theta) = -\sin(\theta)$$
$$\cos(-\theta) = \cos(\theta)$$

根据这些公式，可以用 θ 表示 $\mathbf{M}_z^{-1}$，结果如下：

$$\mathbf{M}_z^{-1} = \begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

下面来计算 $\mathbf{M}_z*\mathbf{M}_z^{-1}$，看能否得到单位矩阵 $\mathbf{I}_{4\times4}$：

$$\mathbf{M}_z*\mathbf{M}_z^{-1} =$$
$$= \underbrace{\begin{vmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}}_{\mathbf{M}_z} * \underbrace{\begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}}_{\mathbf{M}_z^{-1}}$$
$$=$$
$$\begin{vmatrix} (\cos^2\theta+\sin^2\theta) & (-\sin\theta*\cos\theta+\cos\theta*\sin\theta) & (0) & (0) \\ (-\sin\theta*\cos\theta+\cos\theta*\sin\theta) & (\sin^2\theta+\cos^2\theta) & (0) & (0) \\ (0) & (0) & (1) & (0) \\ (0) & (0) & (0) & (1) \end{vmatrix}$$

根据恒等式 $\cos^2\theta+\sin^2\theta=1$，可将上述矩阵简化为：

$$= \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} = \mathbf{I}_{4\times4}.$$

现在知道了根据几何解释，来计算几何变换矩阵的逆矩阵的威力了吧。下面介绍一些数学知识。旋转矩阵是一种特殊的矩阵，实际上，它是一种非常独特的矩阵：正交矩阵。

7. 正交矩阵

正交（Orthonormal）矩阵指的是，在给定的基（basis）的情况下，矩阵的每一行（列）都与前一行（列）垂直。基是一组向量，可用于表示空间中的任何向量。在这里，标准 2D 空间的基是<*i* + *j*>，标准 3D 空间的基是<*i* + *j* + *k*>。在 3D 空间中，所有向量都可以表示为 *i*、*j*、*k* 的线性组合：

$$\mathbf{u} = u_x*\mathbf{i} + u_y*\mathbf{j} + u_z*\mathbf{k}$$

其中 *i* = <1，0，0>，*j* = <0，1，0>，*k* = <0，0，1>，因此，*i*、*j*、*k* 相互垂直。

如果矩阵包含的各行（列）是一组正交向量，则该矩阵为正交矩阵。下面再来看一下矩阵 $\mathbf{M}_z$：

$$\mathbf{M}_z = \begin{vmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

如果假设该矩阵的各行表示的是 3D 向量，则：

$$\mathbf{U1} = \langle \cos\theta, \sin\theta, 0, 0 \rangle$$
$$\mathbf{U2} = \langle -\sin\theta, \cos\theta, 0, 0 \rangle$$
$$\mathbf{U3} = \langle 0, 0, 1, 0 \rangle$$
$$\mathbf{U4} = \langle 0, 0, 0, 1 \rangle$$

从技术上说，我们现在位于 4D 空间中，因为这些变换都基于齐次坐标的。现在暂时不要考虑这一点，看一看这些向量是否相互垂直。

- **u1** 与 **u4** 相互垂直，因为(**u1** . **u4**) = 0。
- 可以很容易看出，**u3** 和 **u4** 与 **u1** 和 **u2** 相互垂直，因为 **u1** 和 **u2** 的最后两个分量都为 0，因此与 u3 或 u4 的积总是为 0，点积同样也为 0。

剩下的问题是，**u1** 与 **u2** 是否相互垂直。来检查一下：

$$\mathbf{u1} = \langle \cos\theta, \sin\theta, 0, 0 \rangle$$
$$\mathbf{u2} = \langle -\sin\theta, \cos\theta, 0, 0 \rangle$$

$$\mathbf{u1} \cdot \mathbf{u2} = (\cos\theta)*(-\sin\theta) + (\sin\theta)*(\cos\theta) + 0 + 0$$
$$= 0.$$

它们的确相互垂直。我们介绍了正交矩阵的定义以及如何判断，但这些对计算逆矩阵有何帮助呢？如果矩阵为正交矩阵，则其转置矩阵和逆矩阵相同。因此，只需计算 $\mathbf{M}_z^t$，便可得到 $\mathbf{M}_z$ 的逆矩阵（$\mathbf{M}_z^{-1}$）。

给定：

$$\mathbf{M}_z = \begin{vmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

通过交换行和列，可以得到其转置矩阵：

$$\mathbf{M}_z^t = \begin{vmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

它的确是 $\mathbf{M}_z^{-1}$！

4.8 基本几何实体

接下来要介绍的主题是基本几何实体，如点、线、平面和曲面，但重点是 3D 图形学领域涉及的各种数学表示和数学运算。例如，所有的多边形都是有边界的平面，因此熟悉平面操作和平面方程将很有帮助。

4.8.1 点

关于点没有太多要说的。图 4.29 显示了 2D 空间中的点 **p1**(x, y) 和 3D 空间中的点 **p2**(x, y, z)。当然，点 **p1** 和 **p2** 的齐次坐标分别为$[x\ y\ w]$和$[x\ y\ z\ w]$。

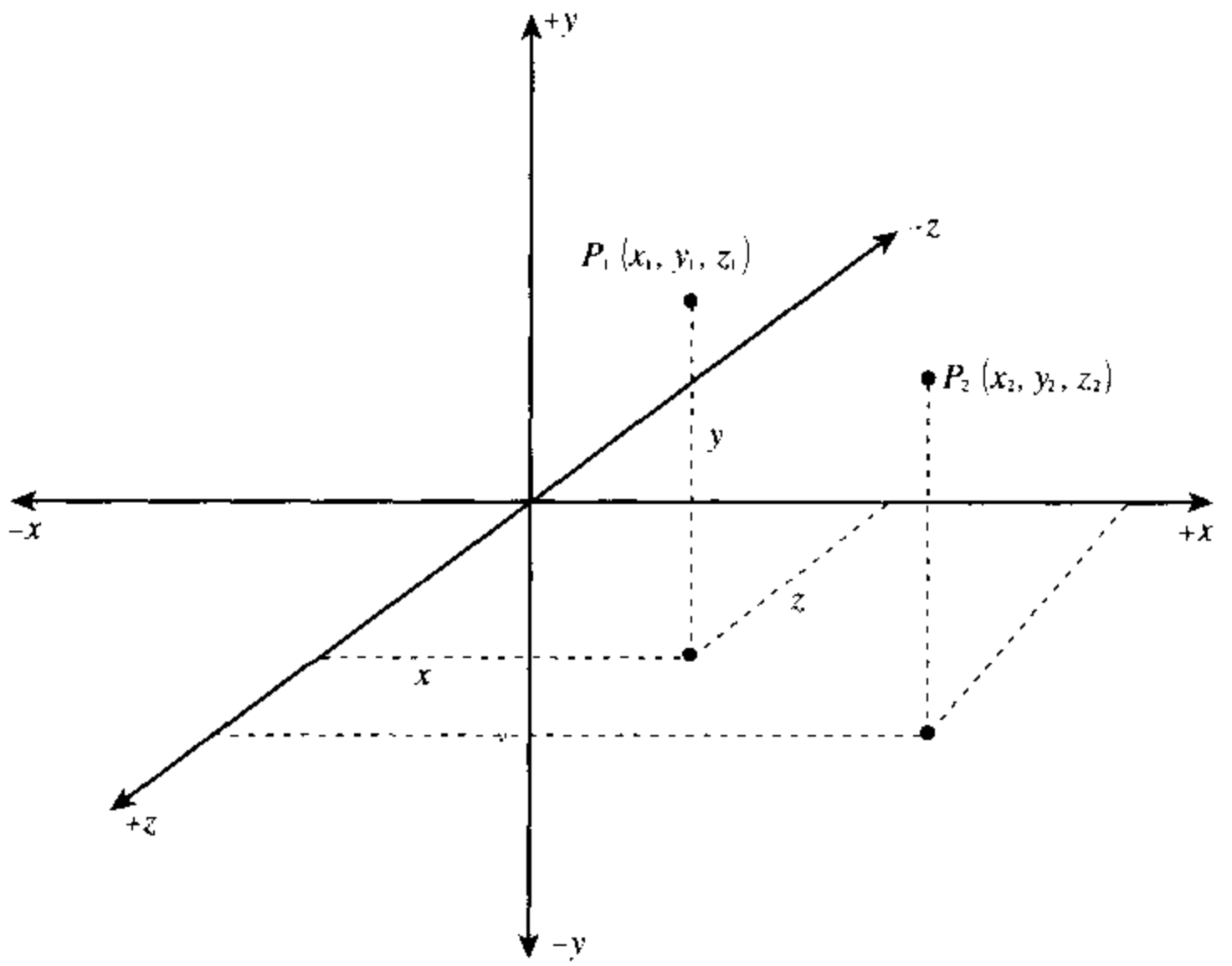


图 4.29 3D 空间中的点

4.8.2 直线

直线更有趣些，且在 3D 空间中表示起来要复杂些。首先来看一下 2D 空间中从点 **p1**$(x1, y1)$ 到 **p2**$(x2, y2)$ 的直线，如图 4.30 所示。这条直线没什么特殊的。表示直线的方法有很多，如点-斜率形式、斜率-截距形式和通用形式。

斜率-截距形式如下：

```
y=m*x + b
```

其中 m 为直线的斜率（dy/dx），b 为 y 轴截距。

点-斜率形式如下：

```
(x-x0) = m*(y-y0)
```

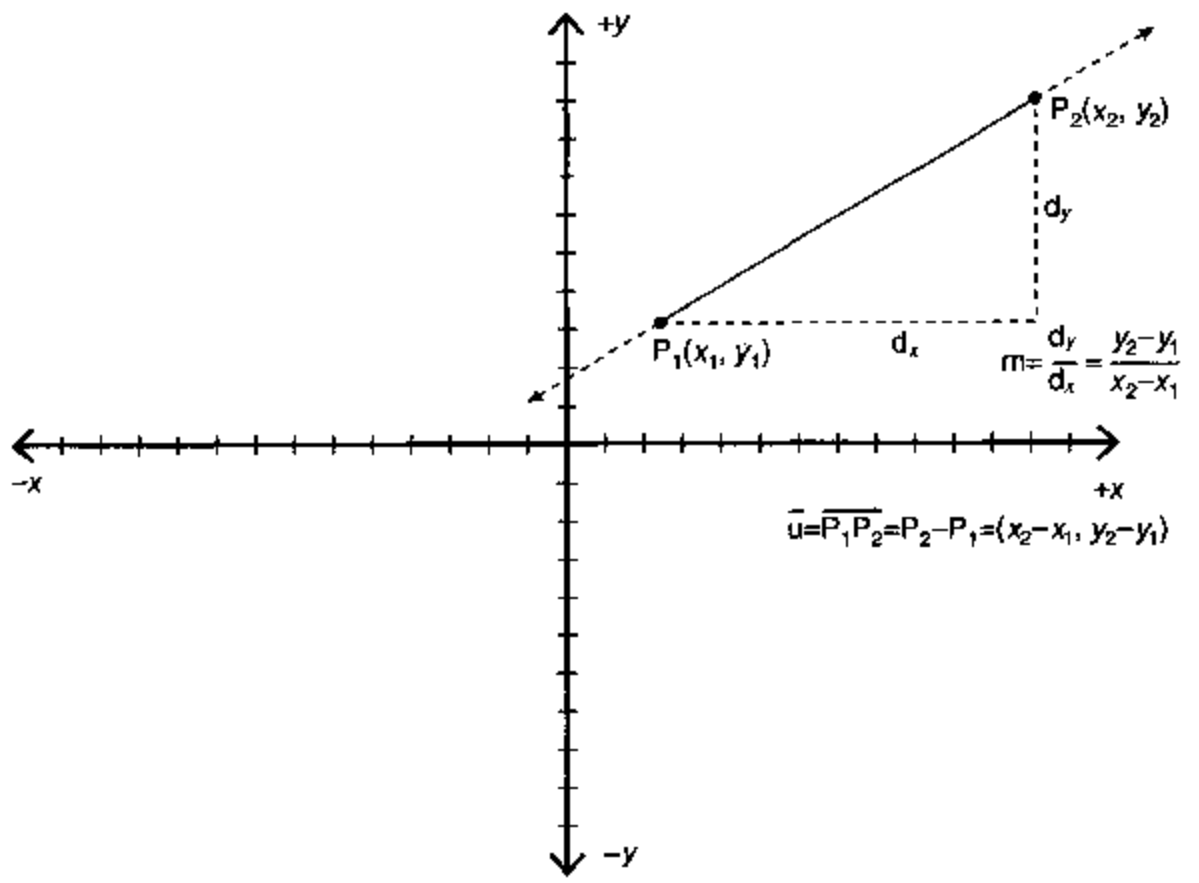


图 4.30　2D 空间中的直线

其中 m 为直线的斜率，(*x*0，*y*0) 为直线上的一个点。

通用形式如下：

```
a*x + b*y + c = 0
```

当然，通用形式是最基本的。下面介绍如何将其他形式转换为通用形式。转换斜率-截距形式很容易：

```
y = m*x + b
```

重新整理后，结果如下：

```
m*x - 1*y + b = 0
```

对照下述通用形式，可以知道系数 *a*、*b*、*c* 的值：

```
a*x + b*y + c = 0
```

因此，*a* = m，*b* = -1，*c* = *b*（截距）。但如果不知道截距该如何办呢？如果只有两点的坐标 (*x*0，*y*0) 和 (*x*1，*y*1)，如何计算系数 *a*、*b*、*c* 呢？来看一下点-斜率形式是否有帮助。首先从点-斜率方程开始：

```
(y - y0) = m*(x - x0)
```

两边乘以 m 后得到：

```
y - y0 = m*x - m*x0
```

将 *x* 和 *y* 放到等式的左边，其他项放到右边：

```
-m*x + y = y0 - m*x0
```

将常量项从等式右边移动到左边，得到：

```
(-m)*x + (1)*y + (-1)*y0 + (m)*x0 = 0
```

从中可知，*a* = −*m*，*b* = 1，*c* = (*m***x*0-*y*0)。一般而言，上述任何一种形式都适用于任何用途。然而，如果要计算两条直线的交点，使用通用形式最合适，因为这样可以将系数放入到矩阵中，然后求解方程组：

```
a1*x + b1*y = c1
a2*x + b2*y = c2
A =    |a1 b1|
       |a2 b2|
```

$$X = \begin{vmatrix} x \\ y \end{vmatrix}$$

$$B = \begin{vmatrix} c1 \\ c2 \end{vmatrix}$$

我们需要求解 **A*X = B**。为此，可以将方程两边分别乘以 **A** 的逆矩阵 $\mathbf{A}^{-1}$，得到：

$$X = A^{-1}*B$$

这在前面已经介绍过，因此将这项任务留给读者去完成。

3D 空间中的直线要复杂些。实际上，很多情况下我们都将使用直线的参数化表示，但后面将把参数化表示作为一个整体进行介绍。现在，来看一下能否找到一种表示 3D 直线的方法。图 4.31 描述了 3D 空间中的一条直线，图中有一条通过点 **p0**(*x0*，*y0*，*z0*)和 **p1**(*x0*，*y0*，*z0*)的直线以及从 p0 到 p1 的向量 $v' = \langle x1\text{-}x0, y1\text{-}y0, z1\text{-}z0\rangle = \langle a, b, c\rangle$。因此我们知道直线上的两个点以及向量 *v'*，后者类似于 2D 空间中直线的斜率。

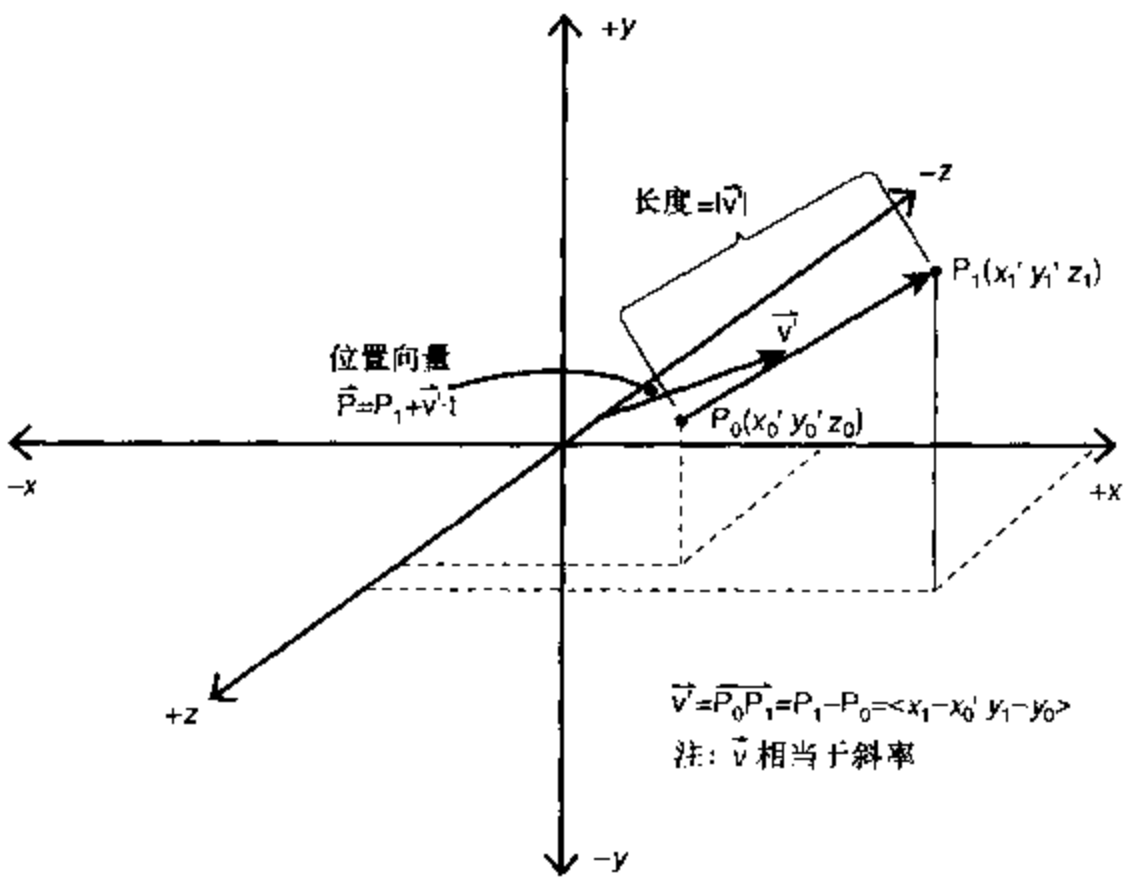


图 4.31　3D 空间中的直线

下面是用向量表示的参数化直线方程：

$$p(x,y,z) = p0 + v'*t$$

当 *t* 从 0 变化到|v|时，**p** 将从 **p0** 移 **p1**。您应该熟悉这些，因为前几节讨论位置和 2D 位移向量时介绍过。3D 直线的显式形式如下：

$$\frac{(x-x0)}{a} = \frac{(y-y0)}{b} = \frac{(z-z0)}{c}$$

这包含三个方程，但您只需要两个，因为它们是耦合的。a、b、c 是单位向量 v'的(*x*，*y*，*z*)分量。这种形式的问题在于有些笨拙。下面来计算从 **p0**(1，2，3)到 **p1**(5，6，7)的 3D 直线的这种形式。

给定：

$$p0(1,2,3)$$
$$p1(5,6,7)$$
$$v = \langle p1-p0\rangle = \langle 5-1,\ 6-2,\ 7-3\rangle = \langle 4,4,4\rangle$$
$$v' = v/|v| = \langle 4,4,4\rangle/|\sqrt{4^2+4^2+4^2}|$$
$$= \langle .57,\ .57,.57\rangle$$

```
= <a,  b,  c>
```

将 a、b、c 代入上述方程，结果如下：

$$\frac{(x-1)}{.57}=\frac{(y-2)}{.57}=\frac{(z-3)}{.57}$$

乘以 0.57，以消除分母，并分离等式，结果如下：

```
(x-1) = (y-2)
(y-2) = (z-3)
```

简化后得到：

```
x - y = -1
y - z = -1
```

在 3D 空间中，显式形式的直线难以理解；参数化形式要清楚得多。

4.8.3　平面

平面是 3D 图形学的关键部分。从技术上说，平面将延伸到无穷远处，如图 4.32 所示。然而，在处理 3D 图形时，我们经常需要用到多边形的概念，它是位于平面内的封闭对象，由很多边（直线）组成，如图 4.33 所示。由于每个多边形通常位于一个平面中，因此可以根据该平面信息得到很多关于多边形的信息；反过来，也可以根据多边形推演出其所在平面的信息，并将其用于数学运算。因此，理解平面是非常重要的。我们首先介绍平面的两个通用概念属性：

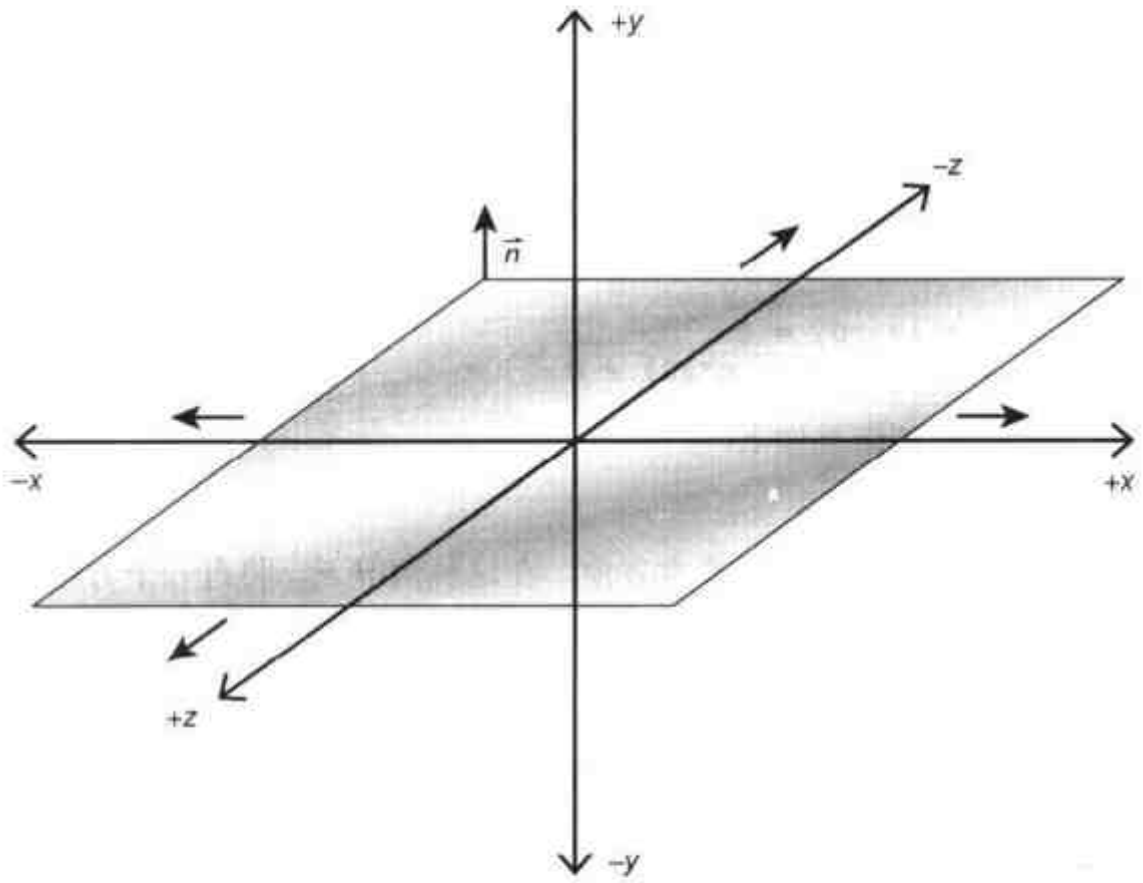


图 4.32　一个无穷大平面

- 所有 3D 平面都延伸到无穷远。
- 所有平面都将整个空间分成两个半空间（这对于各种空间划分算法和碰撞检测非常重要）。

有很多方法可用来生成平面方程，下面从几何学的角度分析这个问题，看能否推导出平面方程。这是一个非常重要的练习——您可以自己推导出一些东西，以后将对您有很大帮助。图 4.33 描述了一个平面，该平面的法线向量 $n=<a, b, c>$，点 **p0**($x0$, $y0$, $z0$) 和 **p**(x, y, z) 位于该平面上。

我们看到了要定义的平面，但其平面方程是什么呢？向量 **p0->p** 位于该平面中，因此与 n 垂直。现在我们有了一个约束条件：对于该平面上的任何一个点 **p**(x, y, z)，向量 **p0->p** 与 **n** 的点积都等于零，因为两个垂直的向量的点积为零。所以：

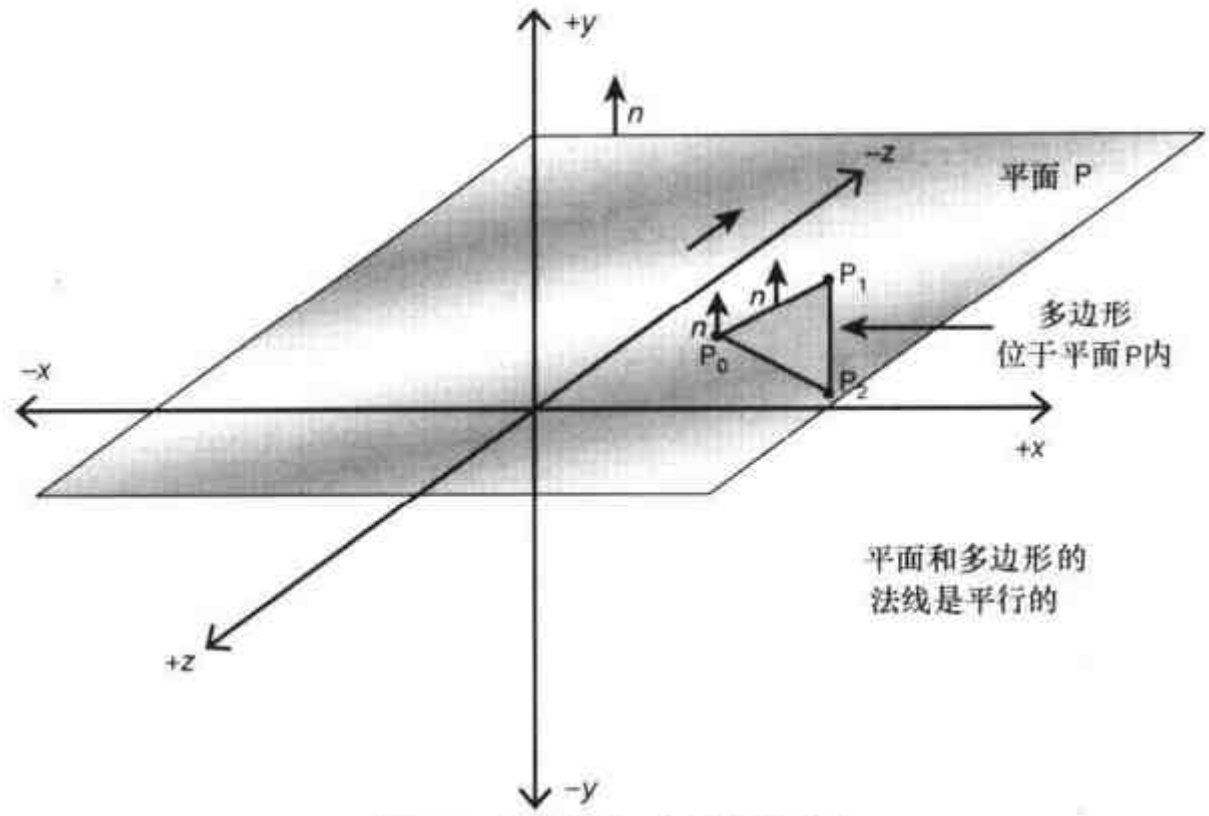


图 4.33　根据多边形创建出的平面

```
n . (p0->p) = 0
```

这就是平面的方程！现在执行乘法，得到如下结果：

```
<a,b,c> . <x-x0, y-y0, z-z0) = 0
```

```
a*(x-x0) + b*(y-y0) + c*(z-z0) = 0
```

这是平面方程的点-法线形式，($x0$，$y0$，$z0$)是平面上的一个点，而<a，b，c>是平面的法线。现在，将各项相乘，并整理常量，得到平面方程的通用形式：

给定

```
a*(x-x0) + b*(y-y0) + c*(z-z0) = 0
a*x + b*y + c*z + (-a*x0 - b*y0 - c*z0) = 0
```

令 d = (-a*$x0$ – b*$y0$ – c*$z0$)，结果如下：

```
a*x + b*y + c*z + d = 0
```

计算两个平面的交线时，使用通用形式非常方便。在这种情况下，将有一个方程组，它由两个或更多关于 x、y、z 的方程组成，您只需求解该方程组即可。

1. 判断点位于哪个半空间中

有一项您将经常使用的、与平面相关的非常重要的操作：判断点位于平面的哪一个半空间中。例如，您可能想知道炮弹是否穿过了平面等。平面方程可以告诉您这一点。在图 4.34 中，有一个平面和一个点，我们要判断这个点位于平面的正半空间还是负半空间中。正半空间是法线指向的空间，而负半空间是另一个半空间。

为判断这一点，假设有如下形式的平面方程：

```
hs = a*(x-x0) + b*(y-y0) + c*(z-z0)
```

只需将点(x，y，z)代入到方程中，并测试结果：

- 如果 hs = 0，该点位于平面上。
- 如果（hs > 0），该点位于平面的正半空间中。
- 如果（hs < 0），该点位于平面的负半空间中。

这是检测点位于多边形哪一边的方法之一。下面来看一个例子，其中测试平面为 x-z 平面。法线向量为<0, 1, 0>，平面上的一个点为(0，0，0)，因此平面方程如下：

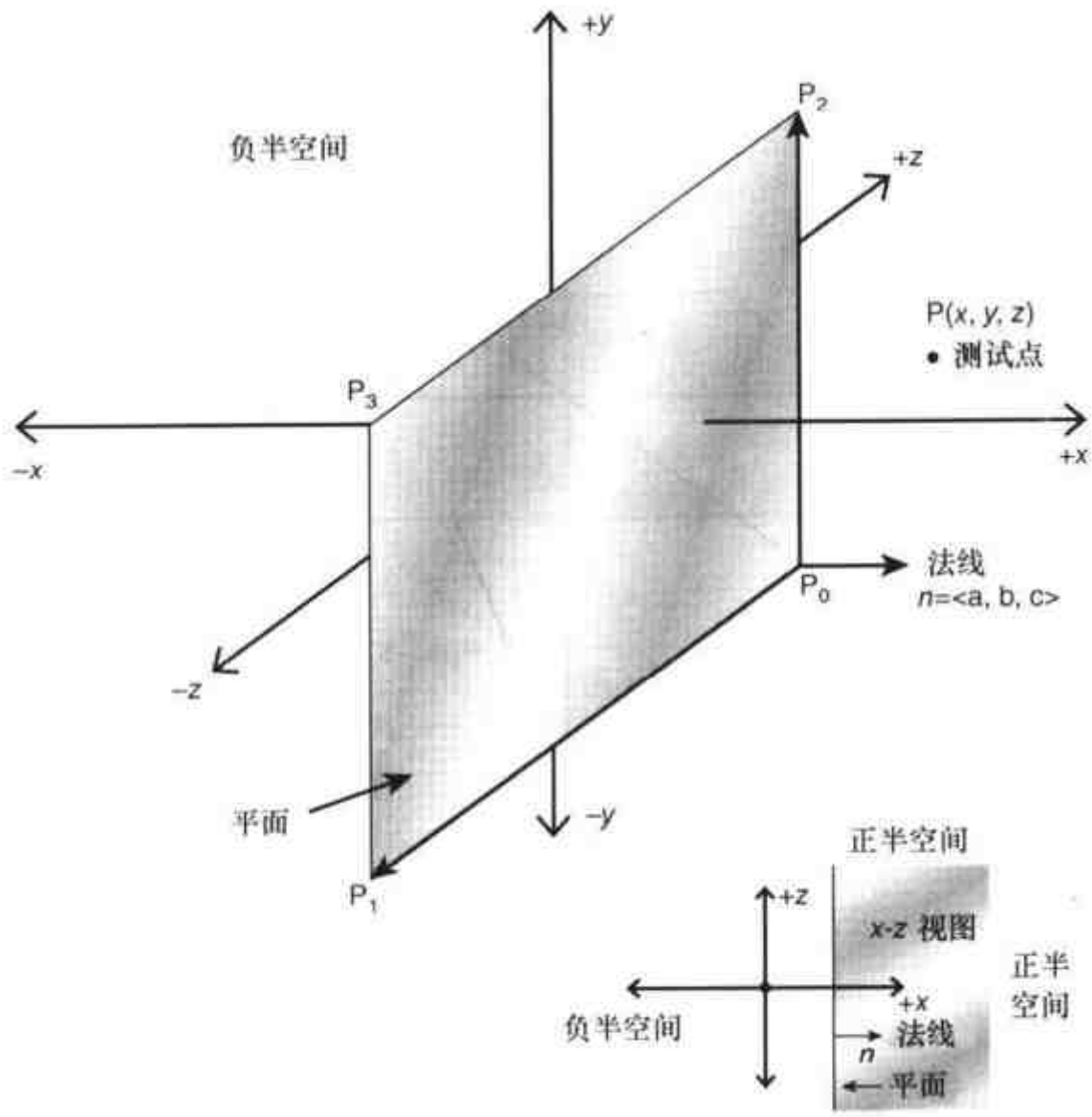


图 4.34　平面及其半空间

```
hs = 0*(x-0) + 1*(y-0) + 0*(z-0) = (y-0)
```

首先 x、z 坐标是无关的，因为测试平面为 *x-z* 平面；因此只需要考虑点的 *y* 坐标。对于点 **p**(10，10，10)，它显然位于正半空间中；因为 $(y-y0)=(10-0)=10>0$，因此该点位于正半空间中。在这种情况下，问题简化为对单个坐标的测试。

2. 使用显式方程计算 3D 空间中平面与直线的交点

下面介绍如何计算 3D 空间中直线与平面的交点。这种计算在 3D 游戏中非常有用，其原因有很多：碰撞检测、裁剪等。我们将使用图 4.35 所示的平面和直线。为简化起见，我选择 x-z 平面，点 **p0**(0，0，0)位于该平面上，平面的法线向量为 n = <0，1，0>；直线通过点 **p1**(4，4，4)和 **p2**(5，-5，-4)。现在，来看能否计算出图 4.35 中直线与平面的交点 $\mathbf{p}_i(x, y, z)$。

平面的点-法线方程如下：

```
a*(x-x0) + b*(y-y0) + c*(z-z0) = 0
```

用法线向量代替<*a*，*b*，*c*>，并将点 **p0**(*x*0，*y*0，*z*0)代入方程，得到如下平面方程：

```
0*(x-0)+1*(y-0)+0*(z-0) = 0
                      y = 0
```

3D 直线方程的通用形式如下：

$$\frac{(x-x0)}{a}=\frac{(y-y0)}{b}=\frac{(z-z0)}{c}$$

其中<a，b，c>是方向与 **p1**->**p2** 相同的单位向量，可以这样计算得到：

```
v = p2-p1 =  (5,-5,-4) - (4,4,4)= <1, -9, -8>
v'= v/|v| = <1,-9,-8>/<sqrt((1)²+(-9)²+(-8)²)
          = <0.08, -0.74, -0.66>
```

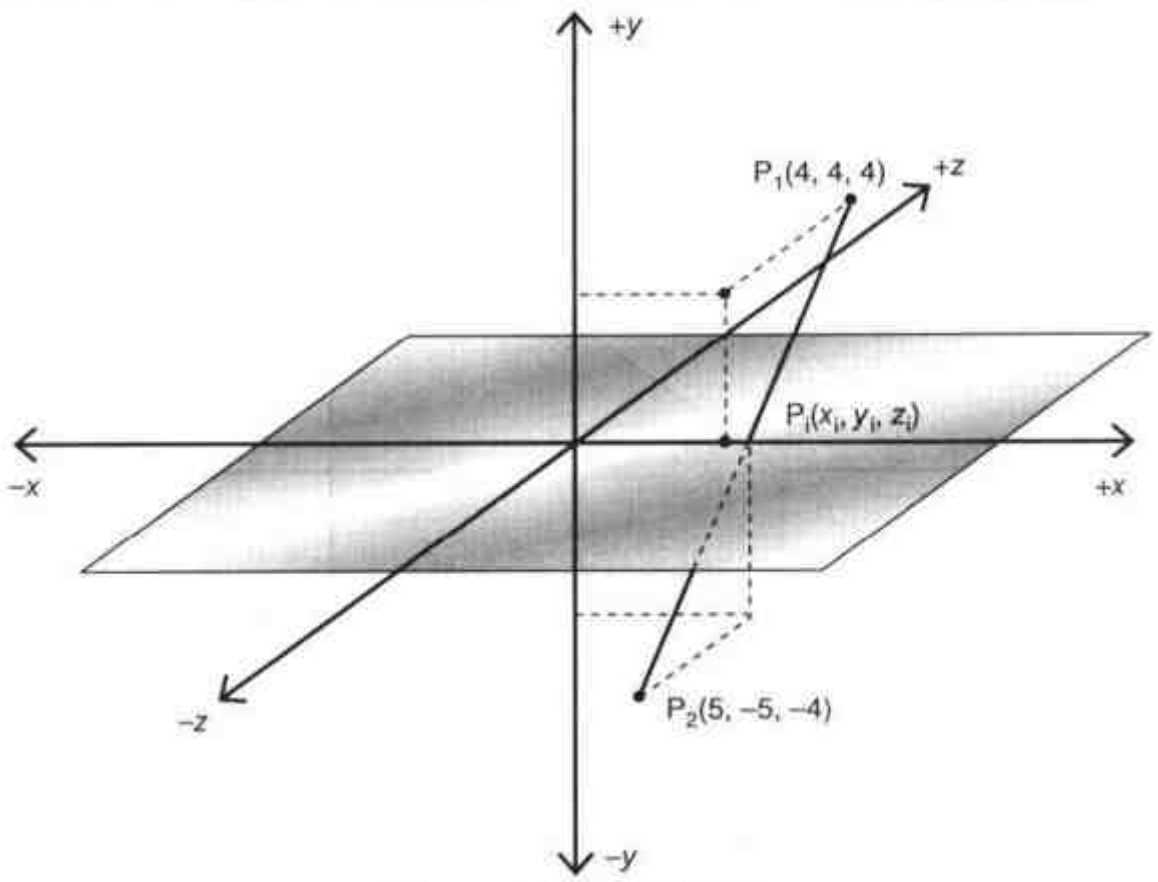


图 4.35　平面与直线的交点

用点 **p1** 替换(*x*0，*y*0，*z*0)，并用 **v** 替换(*a*，*b*，*c*)，结果如下：

$$\frac{(x-4)}{0.08}=\frac{(y-4)}{-0.74}=\frac{(z-4)}{-0.66}$$

整理直线方程和平面方程，结果如下：

(*x*-4)/.08 = -(*y*-4)/.74 = -(*z*-4)/.66（直线方程）

y = 0（平面方程）

现在可以求解该方程组了。将 *y* = 0 代入如下直线方程：

```
 (x-4)/.08 = -(y-4)/.74
-(y-4)/.74 = -(z-4)/.66
```

结果如下：

```
 (x-4)/.08 = -(0-4)/.74
-(0-4)/.74 = -(z-4)/.66
x = 4+(.08)*4/.74 = 4.43
z = 4 -(0.66)*4/.74 = .43
```

因此，直线与平面的交点为 $\mathbf{p}_i$(4.43，0，0.43)。这个范例有些繁琐，但是很重要。后面将根据几何知识，推导出一种更快地计算直线与平面交点的方法。

4.9　使用参数化方程

参数化方程在计算机图形学中很有用，因为通过使用参数化方程，可以用单变量（而不是 2 或 3 个变量）函数来表示直线或曲线。另外，在计算机游戏中，它是一种更自然的表示运动和轨迹的方法。下面详细介绍它。

4.9.1　2D 参数化直线

在图 4.36 中，有两个位于 x-y 平面中的点：**p0**(*x*0，*y*0)和 **p1**(*x*1，*y*1)。我们知道，向量 **v** = <*vx*，*vy*> = **p0**->**p1** = <*x*1-*x*0，*y*1-*y*0>。如果将 **v** 与 **p0** 相加，结果将为 **p1**。现在的问题是，如何以参数形式表示点，使

得参数 t 在闭区间[a, b]中变化时，点将沿方向 v 从 **p0** 移到 **p1**。当然，在有关位置向量的讨论中，介绍过这方面的内容。可以这样描述从 **p0** 到 **p1** 的点集：

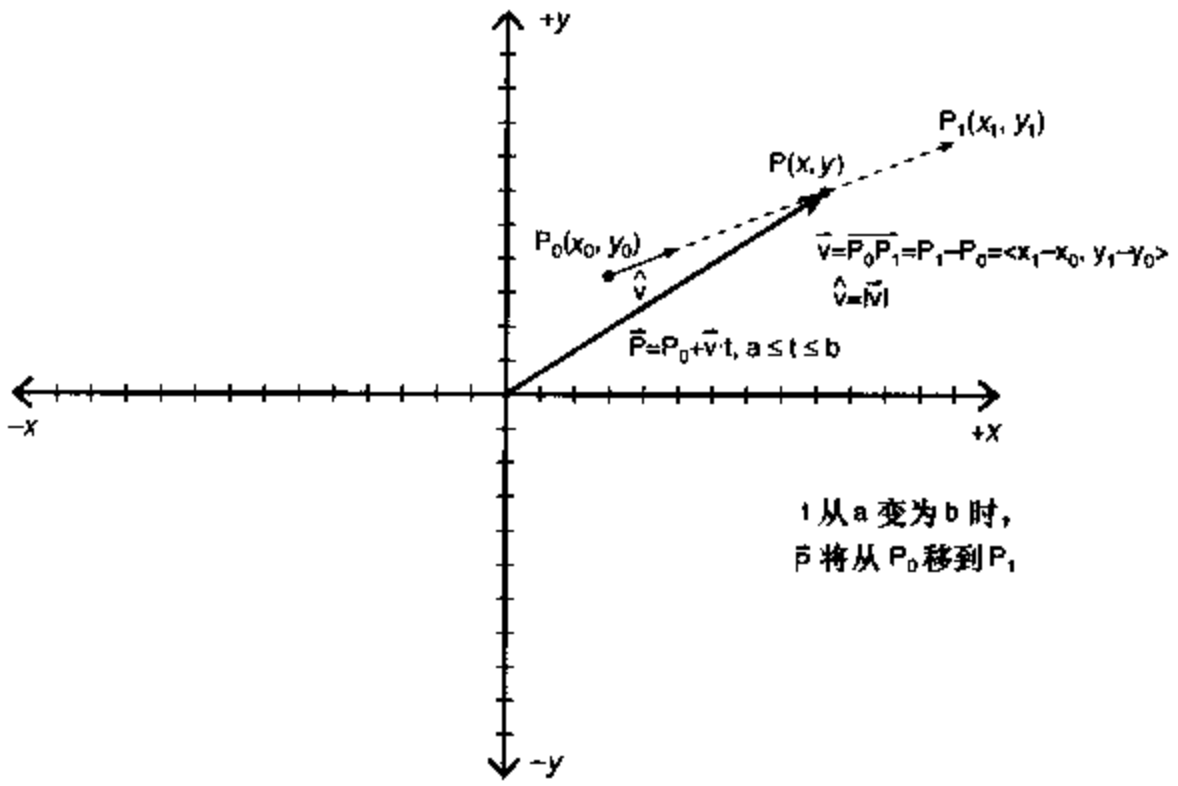


图 4.36　参数化直线

```
p = p0 + v*t
```

其中 **v** = <*vx*，*vy*> = **p0->p1** = (*x*1-*x*0，*y*1-*y*0)；参数 *t* 的取值范围为[*a*，*b*]

这个公式存在的问题是，*t* 的取值区间并不是已知的。如果愿意，可以计算它。这里有一个非常微妙的地方，上述公式是通用的参数化直线方程，直线从-∞到+∞。然而，很多时候，我们需要的是参数化线段，而不是参数化直线。参数化线段与参数化直线的方程相同，但参数 t 的取值范围更小。有两种常见的调整 v 长度的方法，下面分别介绍它们。

1. 用标准方向向量表示的参数化线段

在前面的讨论中，**v** = <*vx*，*vy*> = **p0->p1** = <*x*1-*x*0，*y*1-*y*0>。因此，向量 **v** 的长度为点 **p0** 到 **p1** 的距离，如图 4.36 所示。因此对于从 **p0** 到 **p1** 的线段，参数 t 的变化区间为[0，1]。

注意："[" 和 "]" 表示闭区间，"(" 和 ")" 表示开区间。因此，[0, 1]指的是 0～1 的所有实数，且包括 0 和 1；(0, 1)指的也是 0～1 的所有实数，且包括 1，但不包括 0。

因此参数化线段的方程如下：

```
p = p0 + v*t
```

其中 **v** = <*vx*，*vy*> = **p0->p1** = (*x*1-*x*0，*y*1-*y*0)，t 的取值范围为[0，1]。

t = 0 时，结果如下：

```
p = p0 + v*t = p0
```

这是正确的。

t=1 时，结果如下：

```
p = p0+v*1 = p0+v = p0+<vx, vy> = p1
```

这也是正确的。

因此，参数 *t* 的取值区间[0，1]定义了从 **p0** 到 **p1** 的线段。另外，*t* 的取值也可以在该区间外，但对应的点不在线段上。对于求解参数化方程组来说，这是一个非常重要的属性：如果线段的参数 *t* 的取值区间为[0，1]，但计算得到的 *t* 值不在该区间内，则表明对应的点不在线段上——这是一个非常重要的细节。

另外，使用标准方向向量 v 和区间[0，1]很不错，因为 0 和 1 都是非常特殊的数！下面介绍使用归一化方向向量表示的参数化线段的方法。

2. 使用单位方向向量表示参数化线段

这种参数化直线表示法与前一种方法相同，方向向量被归一化（变成 **v'**）。用归一化向量 **v'**代替 **v** 后，方程变为：

p = p0 + v'*t

其中 **v** = <*vx*，*vy*> = **p0->p1** = <*x*1-*x*0，*y*1-*y*0>，**v'** = **v**/|**v**|。

现在的问题是，什么样的区间定义了从 **p0** 到 **p1** 的线段，如图 4.37 所示？如果您仍然清醒，将很容易看出应该是[0，|**v**|]！下面来验证这一点。

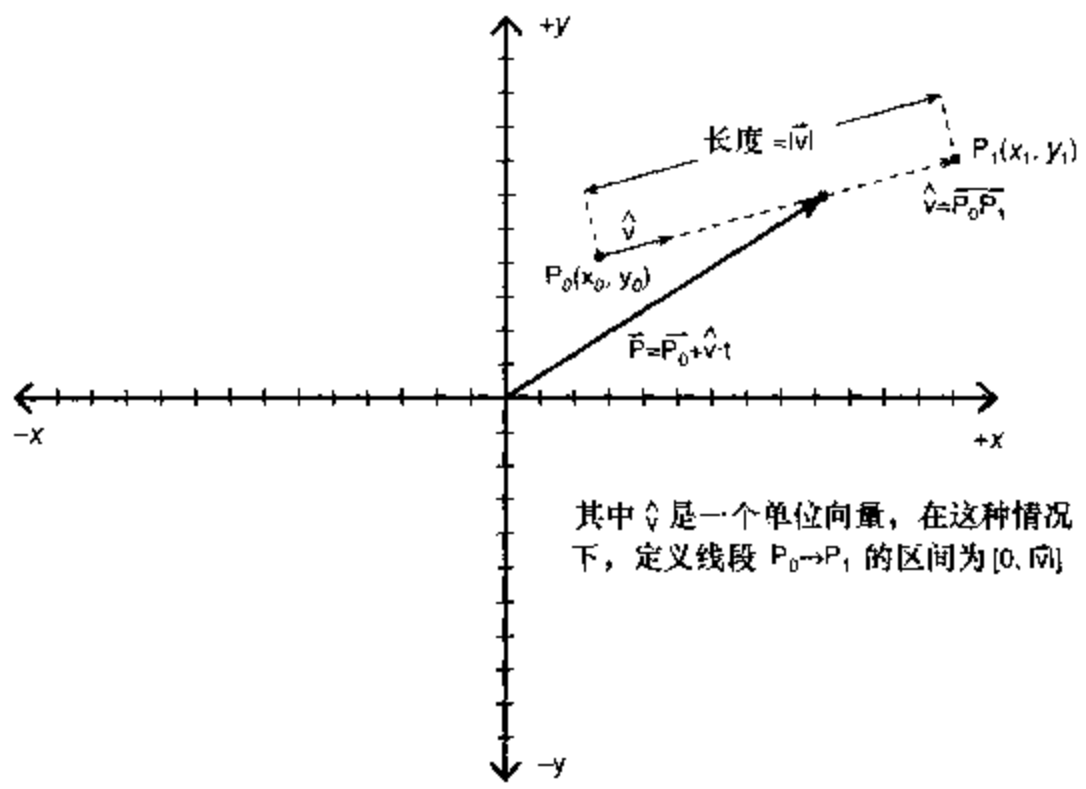


图 4.37　定义从 **p0** 到 **p1** 的线段的区间

t = 0 时：

p = p0 + v'*0 = p0

它是正确的。

t = |v|时：

p = p0 + v'*|v|

$$= p0 + \frac{v}{|v|} * |v|$$

= p0 + v

= p0 + <vx, vy> = p1

这也是正确的。

归根结底，这两种表示方法之间并没有区别，但参数 t 的取值区间不同，前者为[0，1]，后者为[0，|**v**|]。还可以创建这样的参数化方程，它表示从 **p0** 到 **p1** 的线段，t 的取值区间为[-1，1]；这项工作留给读者去完成。

4.9.2　3D 参数化直线

3D 参数化直线与 2D 参数化直线相同：点 **p0**(*x*0，*y*0，*z*0)和 **p1**(*x*1，*y*1，*z*1)都有 3 个分量；当然 **v** 也一样，**v** = <*vx*，*vy*，*vz*> = **p0->p1** = <*x*1-*x*0，*y*1-*y*0，*z*1-*z*0>。向量方程相同：

```
p = p0 + v*t
```

其中

```
v = <vx, vy, vz> ≈ p0->p1 = (x1-x0, y1-y0, z1-z0)
```

除增加一个分量外，其他一切相同：从 2D 变成 3D 非常简单。还记得用 x、y、z 表示的 3D 直线是多么复杂吗？但 3D 参数化直线非常简单。这也是为什么我们在 3D 引擎中要使用参数形式来表示直线的原因，因为显式 3D 直线很难表示，对它执行计算更困难。

1. 计算参数化直线的交点

假设有两艘飞船位于 2D 轨道上，如图 4.38 所示。从中可知：

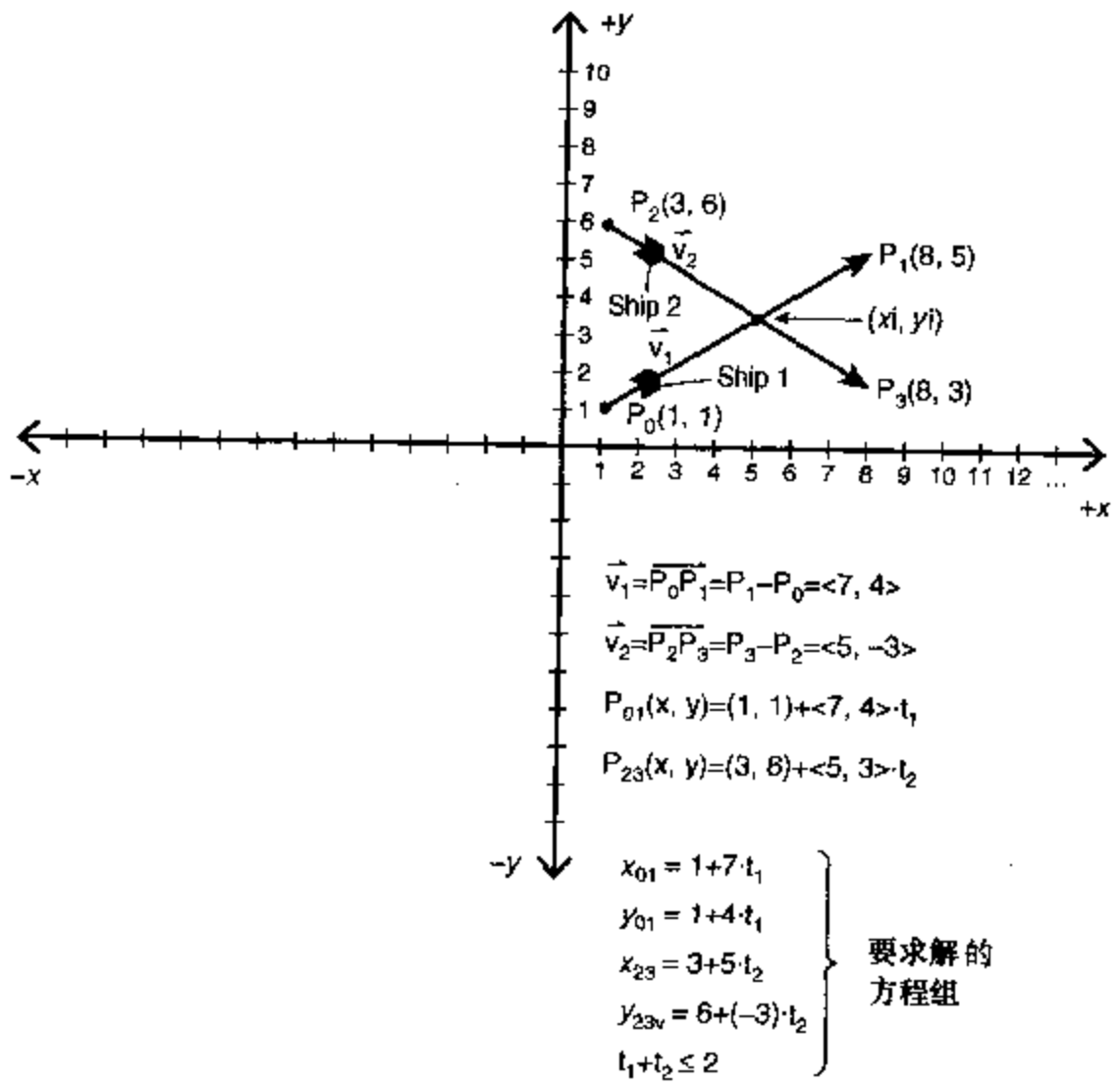


图 4.38　位于 2D 轨道上的两艘飞船

- 飞船 1 从 **p0**($x0$，$y0$) 移到 **p1**($x1$，$y1$)。
- 飞船 2 从 **p2**($x2$，$y2$) 移到 **p3**($x3$，$y3$)。

我们要确定它们的轨道是否相交。您可能首先想为这两条轨道创建两条显式直线，然后检查这两条直线是否相交；但问题是，您不知道交点是否在线段 **p0->p1** 和 **p2->p3** 上。这也是为什么参数化直线如此有用的原因。我们可以执行计算，得到相交点的 t 值。如果 t 不在定义线段的区间中，则表明两条线段并不相交，如图 4.39 所示。

首先来创建两条参数化直线，t 的区间[0，1]，使计算更容易。如果两条直线的 t 区间不同，将相当于从数学上对苹果和桔子进行比较。下面是这两条参数化直线的定义：

已知：

- 飞船 1 从 **p0**（1,1）移动到 **p1**（8,5）。
- 飞船 2 从 **p2**（3,6）移动到 **p3**（8,3）。

计算这两条直线的方向向量：

```
v1 = p0->p1 = <x1-x0, y1-y0> = <8-1, 5-1> = <7,4>
v2 = p2->p3 = <x3-x2, y3-y2> = <8-3, 3-6> = <5,-3>
```

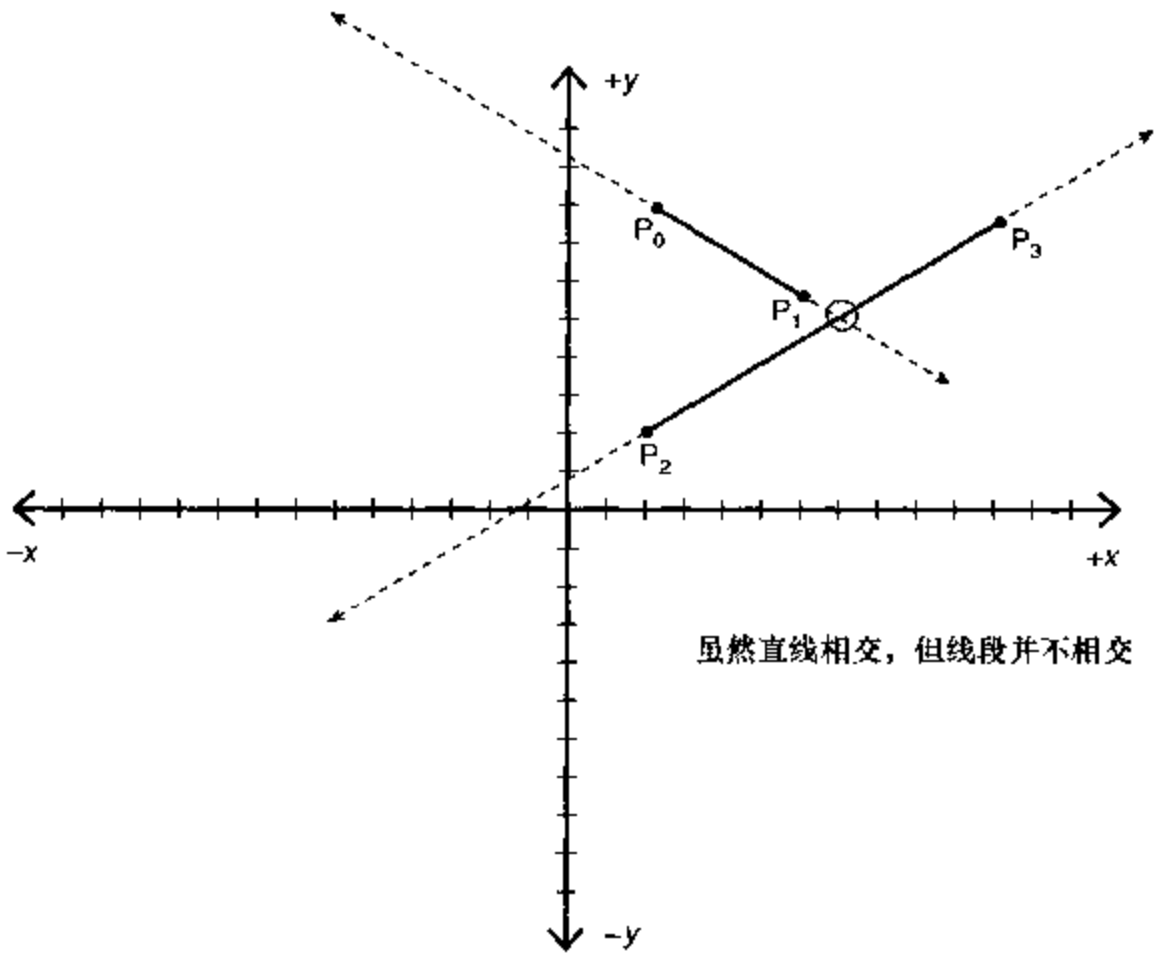


图 4.39　交点必须位于线段上

参数化直线方程如下：

```
p01 = p0 + v1*t1, t1 ∈ [0,1]
p23 = p2 + v2*t2, t2 ∈ [0,1]
```

将实际值代入方程，结果如下：

```
p01(x,y) = (1,1) + <7,4>*t1
p23(x,y) = (3,6) + <5,-3>*t2
```

注意，这两条直线的参数变量不同，我们使用了 $t1$ 和 $t2$，而不是 t，因为这两个参数是不同的。这两个参数都在区间[0，1]内变化，但在交点处两个 t 值可能不同。因此，必须使用不同的参数来跟踪它们——这非常重要，否则将可能导致很多数学问题。

下面写出所有用分量表示的方程：

```
x = 1 + 7*t1
y = 1 + 4*t1

x = 3 + 5*t2
y = 6 - 3*t2
```

我们根据在交点处分量 x 和 y 分别相等，来求解出 $t1$ 和 $t2$，然后检查 $t1$ 和 $t2$ 是否都位于区间[0，1]中。如果是，则两条线段相交，否则没有交点。根据分量相等，可得到如下方程：

```
1 + 7*t1 = 3 + 5*t2
1 + 4*t1 = 6 - 3*t2
```

重新整理后，可得到如下方程：

```
7*t1 - 5*t2 = 2
4*t1 + 3*t2 = 5
```

有两个方程和两个未知数，可以使用任何技术来求解：高斯消元法、克来姆法则、代入法等。结果如下：

```
t1 = .756
t2 = .658
```

两条线段相交，因为 $t1$ 和 $t2$ 都位于区间[0，1]中。要计算交点，可将 $t1$ 或 $t2$ 代入到原来的直线方程

中。下面将 $t1$ 和 $t2$ 都代入到直线方程中，以验证得到的交点坐标（x,y）是否相同。

对于直线 1：

```
x = 1 + 7*t1 = 1 + 7*(.756) = 6.29
y = 1 + 4*t1 = 1 + 4*(.756) = 4.02
```

对于直线 2：

```
x = 3 + 5*t2 = 3 + 5*(.658) = 6.29
y = 6 - 3*t2 = 6 - 3*(.658) = 4.02
```

谢天谢地，完全相同！这些应该足以使您相信直线的参数化表示是最好的，且使用起来非常简单。另外，3D 中的相同问题是绝对相同。没有怪异的公式，只是多了一个坐标分量。在 3D 空间中，将得到类似如下的参数化直线方程：

```
p01(x,y,z) = (x0,y0,z0) + <vx1,vy1,vz1>*t1
p23(x,y,z) = (x2,y2,z2) + <vx2,vy2,vz2>*t2
```

然后将它们分解成三个方程，以求解 $t1$ 和 $t2$：

对于直线 1：

```
x = x0 + vx1*t1
y = y0 + vy1*t1
z = z0 + vz1*t1
```

对于直线 2：

```
x = x2 + vx2*t2
y = y2 + vy2*t2
z = z2 + vz2*t2
```

惟一的差别是，有三个方程和两个未知数，您可以从中选择任何两个方程来求解 $t1$ 和 $t2$。求解方程组后，将 $t1$ 和 $t2$ 代入没有使用的那个方程，以验证结果是否正确。然而，在大多数情况下，并不需要检测一条 3D 线段是否与另一条 3D 线段相交，因为发生这种情况的概率极低。大多数时候，需要检测 3D 参数化直线是否与平面相交。下面介绍这个问题。

2. 计算 3D 参数化直线与 3D 平面的交点

根据平面的不同，计算 3D 直线与平面的交点可能相当简单，也可能非常复杂。如果平面是 x-y、x-z、y-z 或与它们平行的，问题将非常简单。然而，对于通用的直线平面相交问题，需要做更多的工作。

首先写出参数化直线与平面的方程，然后先对最简单的情况求解。下面是 3D 空间中参数化直线的方程：

```
p = p0 + v*t, t ∈ [0,1]

p(x,y,z) = p0(x0,y0,z0) + v<vx, vy, vz>*t
```

下面是 3D 平面方程：

```
n . (p - p0) = 0
nx*(x-x0) + ny*(y-y0) + nz*(z-z0) = 0
```

如图 4.40 所示，我们可以看到三种简单情况下的直线平面相交：

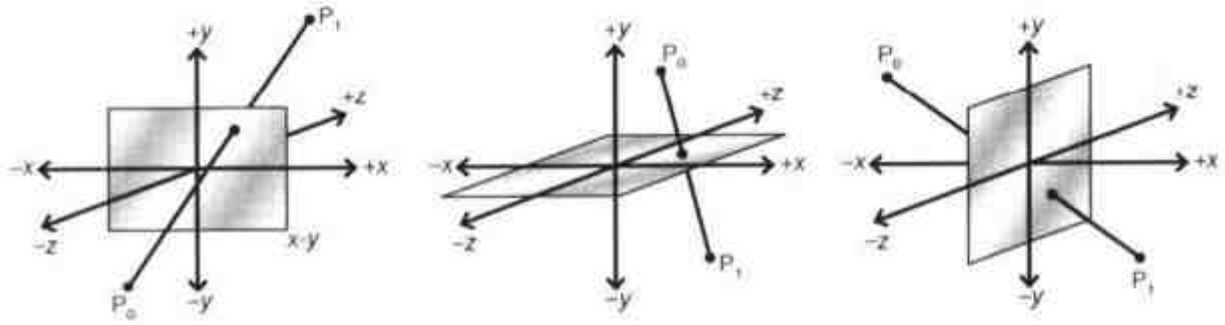
情形 1　直线与 x-y 平面相交（$z=0$）。

情形 2　直线与 x-z 平面相交（$y=0$）。

情形 3　直线与 y-z 平面相交（$x=0$）。

对这三种情形进行求解很简单。我们以 x-y 平面与直线相交为例。在这种情形下，z 坐标 0，将它代入到参数化直线方程中，结果如下：

```
z = z0 + vz*t = 0
t = -z0/vz
```



A. *x-y* 平面　　B. *x-z* 平面　　C. *y-z* 平面

图 4.40　直线与平面的交点（简单情形）

如果 *t* 在区间[0，1]内，则表明两者相交，并可将 *t* 代入分量 *x* 和 *y* 的参数化直线方程中，得到交点的 *x* 和 *y* 坐标：

```
x = x0 + vx*(-z0/vz)
y = y0 + vy*(-z0/vz)
```

在 3D 空间中，直线与任何标准平面或与之平行的平面相交的问题，是相当简单的。问题是如何求得 3D 直线与任意平面的交点。再来看一下参数化直线方程：

```
p(x,y,z) = p0(x0,y0,z0) + v<vx, vy, vz>*t
```

其分量形式为：

```
x = x0 + vx*t
y = y0 + vy*t
z = z0 + vz*t
```

下面是任意平面的方程：

```
n . (p - p0) = 0
nx*(x-x0) + ny*(y-y0) + nz*(z-z0) = 0

nx*x + ny*y + nz*z + (-nx*x0-ny*y0-nz*z0) = 0
```

写成通用形式为：

```
a*x + b*y +c*z + d = 0
```

其中 a = nx，b = ny，c = nz，d =（-nx*x0 – ny*y0 –nz*z0）。

警告：不要混淆了平面的 *x0*、*y0*、*z0* 与直线的 *x0*、*y0*、*z0*，它们是不相同的！这也是为什么我要使用一个符号 *d* 来进行简化的原因。

将参数化直线方程右边的 *x*、*y*、*z* 值代入到通用平面方程中，可以求解出 *t*。如下所示：

```
a*(x0+vx*t) + b*(y0+vy*t) + c*(z0+vz*t) + d = 0
```

求解 t，结果如下：

```
t = -(a*x0 + b*y0 + c*z0 + d)/(a*vx + b*vy + c*vz)
```

其中 *x0*、*y0*、*z0*、*vx*、*vy*、*vz* 来自参数化直线，*a*、*b*、*c*、*d* 来自平面方程。
将 *t* 代入参数化直线方程，可得到交点的 *x*、*y*、*z* 坐标：

```
x = x0 + vx*t
y = y0 + vy*t
z = z0 + vz*t
```

有 *t* 后，只需检测它是否位于区间[0，1]中，便可知道线段与平面相交。

至此，您已经具备了处理 3D 碰撞检测、墙面跟踪、裁剪、武器弹道等所需的数学知识。本书后面将使用这些知识，因此您务必理解它们。

4.10　四元数简介

在 80 年代初，作者与另一位游戏程序员聊到 Atar 的 Space Duel 速度为什么这么快。其处理 3D 速度令人难以置信。这位朋友了解内情，Atari 使用的是矩阵乘法。我对此印象非常深刻，因为在 8 位处理器上执行数学运算需要很高的技巧。现在，我们可以使用 BSP 树、入口、A-缓存等，但如果您不了解四元数将寸步难行。

四元数是由数学家 William Rowan Hamilton 于 19 世纪发明的（如果您学习过图论，应熟悉 Hamiltonian 路径）。四元数显然不是专门为 3D 图形学设计的，但在 3D 图形学的很多方面非常适合使用它：

- 3D 相机控制；
- 压缩存储；
- 平滑 3D 插值。

四元数是基于复数的，理解起来有一点抽象。当然，四元数并不代表现实世界中的任何东西，只在数学意义上存在。然而，复数是一种可用来表示实数无法表达的数学思想的工具。这里首先简要介绍基本的复数理论，再介绍四元数及其数学性质，然后讨论如何将其用于计算机图形学和游戏中。

4.10.1　复数理论

实数集 $\boldsymbol{R}$ 由区间[-∞,+∞]中的所有数组成。来看一些方程：

```
x = sqrt(4) = 2
x = sqrt(1) = 1
x = sqrt(-1) = ???
```

第三个方程是个问题：在实数集中没有-1 的平方根，因为没有哪个实数的平方等于-1。我们需要一个新的数，虚数 i。我们将制定如下规则：

令 i = sqrt（-1）

则 i*i = -1

现在可以计算 sqrt（-4）了：

```
sqrt(-4) = 2*i
```

因为：

$(2*i)*(2*i) = 4*i^2 = 4*-1 = -4$

可以将虚数 i 视为变量或系数，即常规的代数法则仍适用。只需将 i 当成一个与 x 类似的变量，并对其执行任何运算。例如，请看下面的加法：

$3 + 5*i + 3*i^2 - 10 + 6*i$

整理后结果如下：

$= 3*i^2 + 5*i + 6*i + 3 - 10$
$= 3*i^2 + 11*i - 7$
$= 3*(-1) + 11*i - 7$
$= -3 + 11*i - 7$
$= -10 + 11*i$

没什么特别之处。然而，虚数本身非常单调，因此数学家提出了复数的概念，它是一个实数与一个虚数的和。在数学上，复数像这样：

```
z = (a + b*i)
```

其中 a 是实部，b 是虚部。由于实部和虚部不能合并，因此可以将复数视为复平面中的点，如图 4.41 所示。

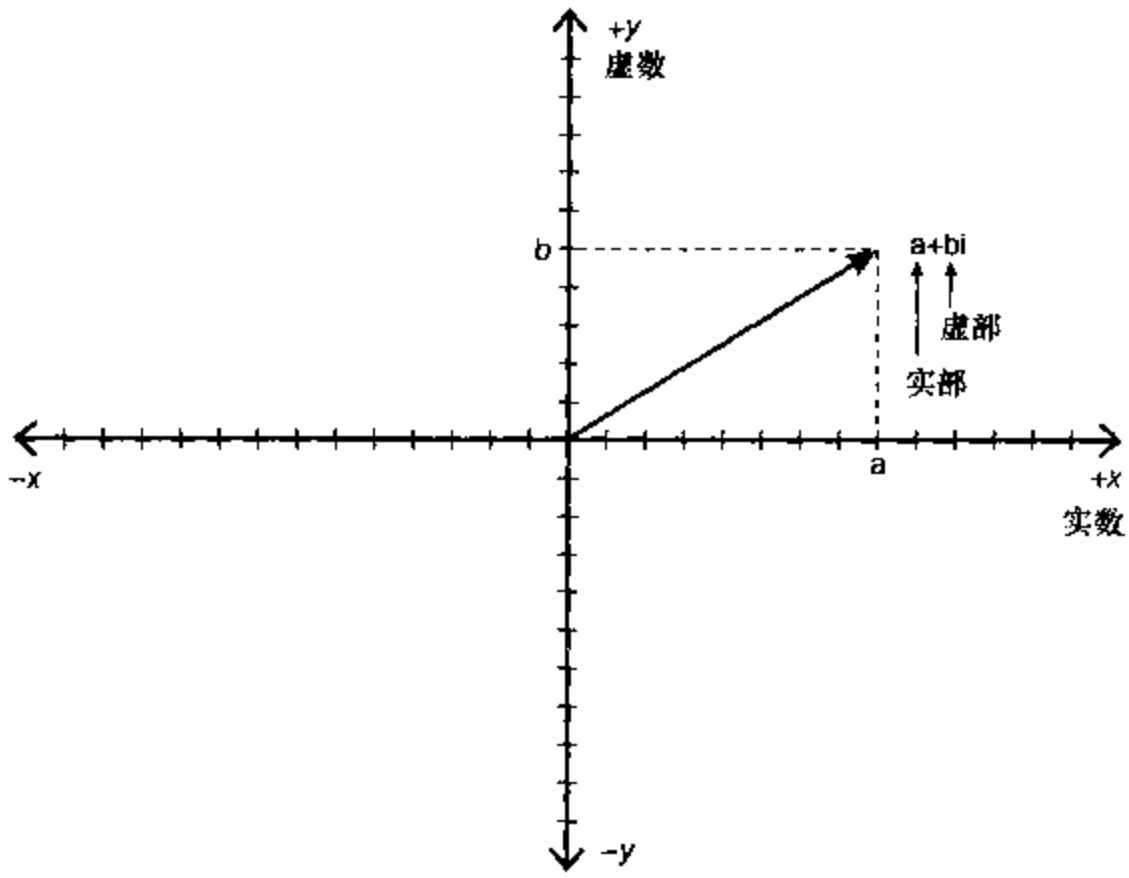


图 4.41　复平面

作为一种约定，大多数人将 x 轴称为实部，将 y 轴称为虚部。因此，可以以向量为基础，得到一种复数的几何表示：

```
z = a*(1,0) + b*(0,i)
```

有关这种概念稍后再介绍，现在读者只需记住它即可。下面来看一下复数运算，如何执行加法、减法、乘法、除法等。

1. 复数与标量相乘/相除

将标量与复数相乘只需对复数的各个分量执行相应的运算即可：
给定：

$z_1 = (a+b*i)$
$k*z_1 = k*(a+b*i) = (k*a+(k*b)*i)$

范例：

```
3*(2+5*i) = (6+15*i)
```

也就是说，将标量分别乘以复数的各个分量。除法与此类似，因为可以将除法视为乘以标量的倒数。

2. 复数加法和减法

要将复数相加或相减，只需将实部和虚部分别相加或相减：
给定：

$z_1 = (a+b*i)$
$z_2 = (c+d*i)$
$z_1 + z_2 = (a+b*i) + (c+d*i) = ((a+c) + (b+d)*i)$

范例：

```
(3 + 5*i) + (12 - 2*i) = (15 + 3*i)
```

3. 复数加法恒等元

加法恒等元（additive identity，将其与任何复数相加时，结果为仍为该复数）是（0 + 0*I），因为：

```
(0+0*i) + (a+b*i) = (0+a + (0+b)*i) = (a+b*i)
```

4. 复数加法逆元素

复数加法逆元素（complex additive reverse，任何复数与其加法逆元素相加时，结果为复数恒等元（0 + 0*i））是 $z^* = (-a - b*i)$，因为：

```
(a + b*i) + (-a - b*i) = (a-a) + (b-b)*i) = (0 + 0*i)
```

5. 复数乘法

复数乘法很容易，下面来执行复数乘法：

$z_1 = (a+b*i)$
$z_2 = (c+d*i)$

$z_1*z_2 = (a+b*i) * (c+d*i)$
$= (a*c + a*d*i + b*c*i + b*d*i^2)$

$= (a*c + (a*d+b*c)*i + b*d*(-1))$
$= ((a*c-b*d) + (a*d+b*c)*i)$

因此，实部为(a*c – b*d)，虚部为(a*d + b*c)。下面是一个范例：

```
(1+2*i) * (-3 + 3*i) = ((-3-6) + (-6+3)*i) = (-9 - 3*i)
```

6. 复数除法

复数除法可以通过直接执行除法来完成。例如，可以按如下方法来计算两个复数的商：

$z_1 = (a+b*i)$
$z_2 = (c+d*i)$

$$z_1/z_2 = \frac{(a+b*i)}{(c+d*i)}$$

如果 c = 0 或 d = 0，则除法比较简单；但如果 c 和 d 都不等于 0，问题将有些复杂。问题是如何执行除法运算，以得到形式为(a + b*i)的结果。其技巧在于先将分母转化为标量，这样可以用它分别去除分子的实部和虚部，从而消除分母。要将分母转化为标量，必须乘以其共轭复数（complex conjugate），通常用星号上标（*）表示。

复数 z = (a + b*i) 的共轭复数是 $z^* = (a - b*i)$。将复数乘以其共轭复数时，结果总是一个实数。

给定：

```
z = (a+b*i)
```

则：

$z * z^* =$
$(a+b*i) * (a-b*i) = (a^2 + a*b*i - a*b*i - b^2*i^2)$
$= (a^2 + b^2)$

使用这种技巧，可以将商问题转换为更好的形式。

给定两个复数的商：

$$\frac{(a+b*i)}{(c+b*i)}$$

总是可以乘以 1，而不会改变其结果，因此对分子分母同时乘以分母的共轭复数：

$$\frac{(a+b*i)}{(c+d*i)} * \frac{(c-d*i)}{(c-d*i)}$$

经过很多步运算后，结果如下：

$$= \frac{(a*c+b*d)}{(a^2+b^2)} + \frac{((b*c-a*d)*i)}{(a^2+b^2)}$$

虽然很难看，但确实为(a + b*i)形式。

7. 倒数

最后一个数学属性（复数必须是封闭集）是倒数。它是这样一个复数，与另一个复数相乘时，结果为“1”：复数中的“1”指的是(1 + 0*i)。可以使用共轭复数来计算倒数。

复数 z =(a + b*i)的倒数如下：

$$1/z = \frac{1}{(a+b*i)}$$

但问题是，这不是（实部 + 虚部*i）形式，需要对其进行整理，使之变成(a + b*I)形式。为此可以乘以共轭复数，结果如下：

$$\frac{1}{(a+b*i)}\ \frac{(a-b*i)}{(a-b*i)} = \frac{a}{(a^2+b^2)} + \frac{b}{(a^2+b^2)}*i$$

8. 复数的向量表示

现在再来看一下复数的表示，介绍另一种看待复数的方式。可以将复数视为 2D 平面中的向量。图 4.41 用 2D 迪卡尔平面来表示复数，其中实部被用作 x 坐标，虚部被用作 y 坐标。因此，没有理由不能将复数视为向量：

```
z = a*(1,0) + b*(0,i)
```

这可简化为：

```
z = <a,b>
```

其中 a 为实部，b 为虚部。图 4.42 说明了这种以向量形式表示复数的方法。

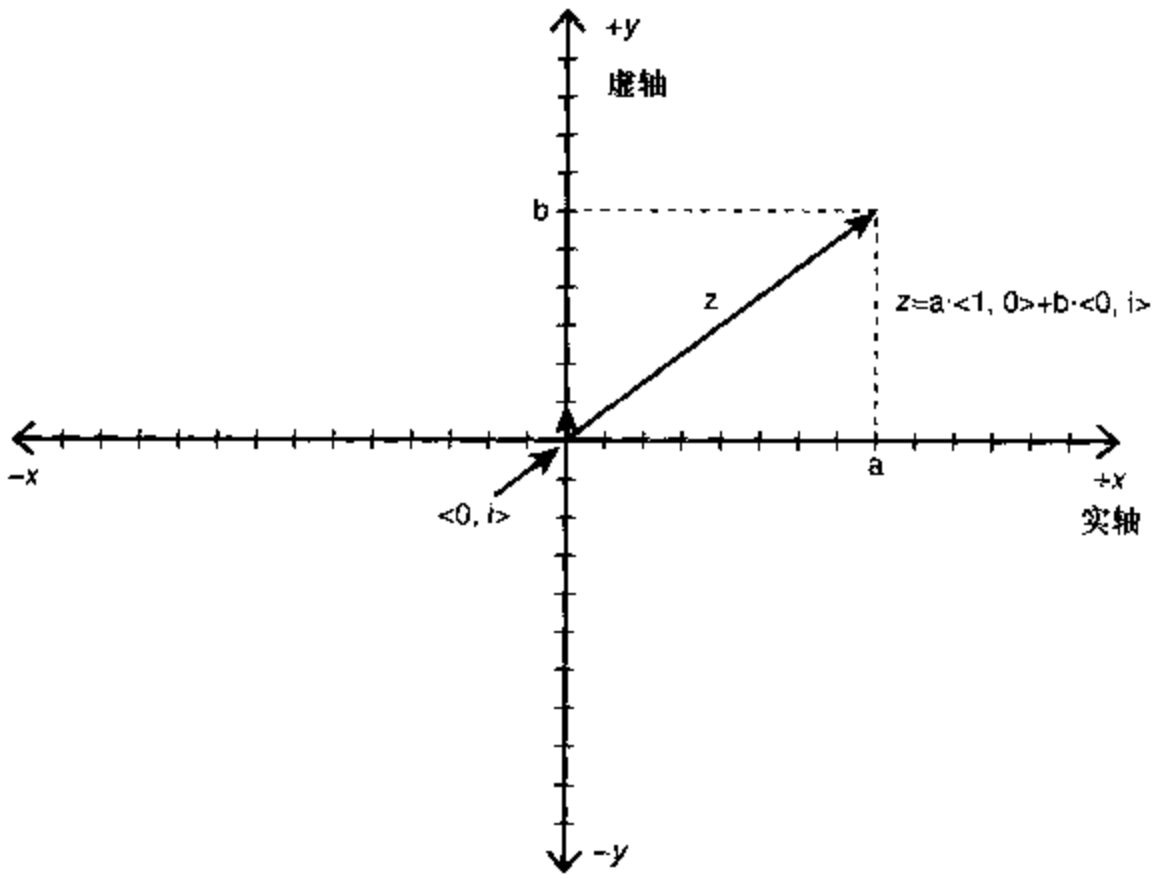


图 4.42 用向量表示复数

将复数表示为向量的优点是，这样可以像向量一样对它们进行变换。另外，将复数表示为向量还让我们能够直观地了解复数，并“看到”采用数学形式无法看到的关系。

9. 复数的范数

很多时候，需要知道复数的长度。当然，从数学角度看，这没有意义，然而将复数视为空间中的向量后，这就很自然，也很容易理解。计算复数范数（norm）的方法如下。

给定：

$z = a+b*i$

则：

$|z| = \sqrt{a^2+b^2}$

下面来看一下复数与其共轭复数的积。

给定：

$z = a+b*i,\ z^* = a-b*i$

则：

$z * z^* = a^2 + b*i - b*i + b^2 = a^2+b^2$

因此，范数也等于：

$|z| = \sqrt{z * z^*}$

4.10.2　超复数

四元数就是超复数（hyper-complex number）。从数学上说，术语超复数可以指任何东西，但通常指有多个虚部的复数。在本书中，超复数是指有一个实部和三个虚部的复数，也叫四元数。

可以用很多种方法来表示四元数，但通常用下述方式来表示：

$q = q_0 + q_1*i + q_2*j + q_3*k$

或

$q = q_0 + q_v$，其中 $q_v = q_1*i + q_2*j + q_3*k$

i=<1，0，0>，j=<0，1，0>，k=<0，0，1>，且 q_0、q_1、q_2、q_3 都是实数。i、j、k 都是虚数，它们组成四元数 q 的向量基（basis）。

虚数基<i，j，k>具有一些有趣的性质。可以将它们视为虚拟坐标系中三个相互垂直的单位向量，用于定义<i，j，k>空间中的点，如图 4.43 所示。然而，关于<i，j，k>最有趣的属性是下面的关系：

$i^2 = j^2 = k^2 = -1 = i*j*k$

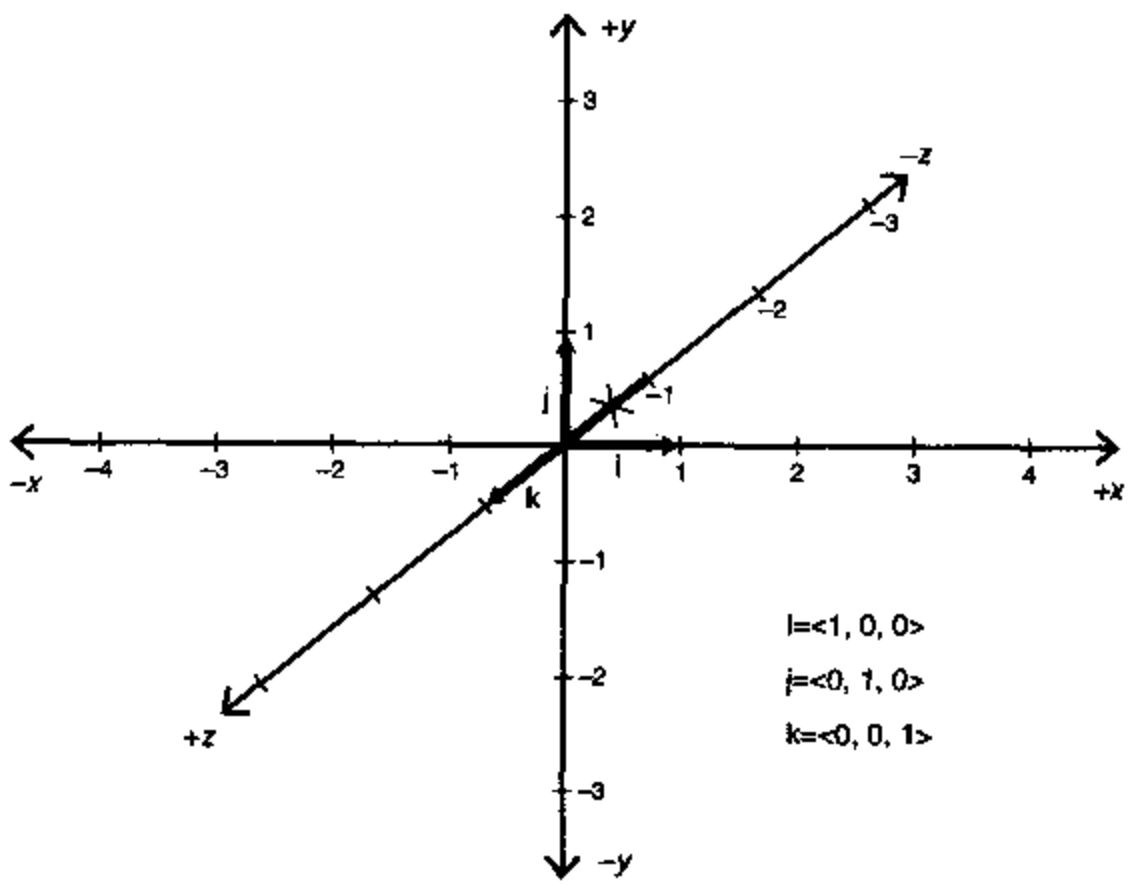


图 4.43　将超复数基（basis）解释为 3D 坐标系

注意：在这里用粗体来显示 *i*、*j*、*k*；可以认为它们是向量，它们也确实是向量，但也可以将它们视为变量。它们有双重性质，有时候适合将它们视为向量，有时候适合将它们视为变量。我可能来回使用其不同的性质，尤其是单独使用 *i*、*j*、*k* 或指出它们之间的关系时。

其中“*=i*j*k*”部分需要花些时间适应，但确实是正确的。当然，可以对上述关系进行变换，如下所示。

给定：

$$i^2 = j^2 = k^2 = -1 = i*j*k$$

因此：

```
i = j*k = -k*j
j = k*i = -i*k
k = i*j = -j*i
```

可以很直观地看出这些公式的正确性，因为任何两个相互垂直的向量的叉积应该与这两个向量本身相垂直。如果交换乘法的顺序，将得到相反的结果（加上负号）。

现在，制定一些四元数书写规则，否则将无法统一四元数表示法。很多人使用小写字母来表示四元数，但也将四元数分成实部和虚部，并将虚部表示为如下形式的向量：

$$\mathbf{q} = q_0 + \mathbf{q}_v$$

其中：

$$\mathbf{q}_v = q_1*\mathbf{i} + q_2*\mathbf{j} + q_3*\mathbf{k}$$

因此 q_0 是实部，$\mathbf{q}_v$ 是虚部，后者是向量$<q_1, q_2, q_3>$。我们将使用这样一种表示法：用粗体小写字母来表示四元数，向量部分也用粗体小写字母表示。下面是一个范例：

$$\mathbf{a} = -1 + 3*\mathbf{i} + 4*\mathbf{j} + 5*\mathbf{k}$$

纯向量形式为：

$$\mathbf{a} = <-1, 3, 4, 5>$$

实部-向量形式为：

$$\mathbf{a} = q_0 + \mathbf{q}_v$$

其中 $q_0 = -1$，并且 $\mathbf{q}_v = <3,4,5> = 3*\mathbf{i} + 4*\mathbf{j} + 5*\mathbf{k}$。

在实际应用中，我们将使用数组来表示四元数，其中第一个数为实部，其他三个数分别为<i，j，k>的虚系数。从现在开始，我将使用下面这种格式来表示四元数：

$$\mathbf{q} = q_0 + x*\mathbf{i} + y*\mathbf{j} + z*\mathbf{k}$$

或用另一种方式：

$$\mathbf{q} = q_0 + <x,y,z> \cdot <\mathbf{i},\mathbf{j},\mathbf{k}>$$

注意：这里利用了*<i, j, k>*的向量性质。

也就是说，将实部表示为 q_0，将虚部表示为 $\mathbf{q}_v = <x, y, z>$，这样可以更容易地将四元数与 3D 空间关联起来。然而，根据需要完成的工作，可能采用其他的形式。

四元数（或任何超复数系统）最重要的性质是，加法、乘法、倒数等运算都与复数理论相同，只是元素更多而已。因此，我们实际上已经知道如何执行这些运算，但需要考虑的是，现在有三个虚数分量，而在基本复数中，只有一个虚数分量。

下面介绍需要对四元数执行的一些基本运算，我将以较快的速度介绍，因为这些运算都是显而易见的。

1. 四元数加法和减法

与常规复数一样，四元数的加法和减法可分别将实部和虚部相加或相减来完成。

范例：

$$q = q_0 + q_v$$
$$p = p_0 + p_v$$

$$q + p = (q_0+p_0) + (q_v+p_v)$$

范例：

$$q = 3 + 4*i + 5*j + 6*k = <3,4,5,6> \text{ in vector form.}$$
$$p = -5 + 2*i + 2*j - 3*k = <-5,2,2,-3> \text{ in vector form.}$$

$$q + p = (3+ -5) + ((4+2)*i + (5+2)*j + (6+ -3)*k)$$
$$=-2 + 6*i + 7+j + 3*k$$

正如您看到的，书写虚数系数很繁琐，可以将它们简写为：

$$<3,4,5,6> + <-5,2,2,-3> = <-2,6,7, 3>.$$

然而，必须特别小心，因为虽然它对于加法和减法是非常合适的，但执行乘法时，必须记住后三个分量是复数。为避免过渡简化，需要这样一种表示四元数的形式：至少将实部与虚部分开。

2. 加法逆元素和加法恒等元

四元数 q 的加法逆元素是这样一个数：将它与 q 相加时，结果为零；它就是-q：
给定：

$$q = q_0 + q_v$$

加法逆元素为：

$$-q = -q_0 - q_v$$

因为：

$$q-q = (q_0-q_0)+ (q_v-q_v) = 0 + 0*i + 0*j + 0*j$$

加法恒等元（或四元数数学中的“0”）为：

$$q = 0 + 0*i + 0*j + 0*k = <0,0,0,0>$$

3. 四元数乘法

加法和减法总是很容易，但乘法总是很复杂！对于四元数也不例外。由于四元数是基于实数和虚数系数的超复数，我们应该总是可以将它们相乘，在乘积中考虑虚数系数，并跟踪它们。下面来执行四元数的乘法。

给定：

$$p = p_0 + p_1*i + p_2*j + p_3*k = p_0 + p_v$$
$$q = q_0 + q_1*i + q_2*j + q_3*k = q_0 + q_v$$

则：

$$p*q = (p_0 + p_1*i + p_2*j + p_3*k) * (q_0 + q_1*i + q_2*j + q_3*k)$$
$$= p_0*q_0 +$$
$$p_0*q_1*i + p_1*q_2*j + p_2*q_3*k +$$
$$p_1*i*q_0 + p_1*i*q_1*i + p_1*i*q_2*j + p_1*i*q_3*k +$$
$$p_2*j*q_0 + p_2*j*q_1*i + p_2*j*q_2*j + p_2*j*q_3*k +$$
$$p_3*k*q_0 + p_3*k*q_1*i + p_3*k*q_2*j + p_3*k*q_3*k$$

如果您仔细观察，将会看到这个乘积中的一些结构——有些是叉积，有些是点积。整理系数，并使用前面的恒等式将虚部简化为：

$$= p_0*q_0 +$$
$$p_0*q_1*i + p_1*q_2*j + p_2*q_3*k +$$
$$p_1*q_0*i + p_1*q_1*i^2 + p_1*i*q_2*j + p_1*i*q_3*k +$$
$$p_2*j*q_0 + p_2*q_1*j*i + p_2*q_2*j^2 + p_2*q_3*j*k +$$
$$p_3*q_0*k + p_3*q_1*k*i + p_3*q_2*k*j + p_3*q_3*k^2$$

现在，有很多方法可以对这些乘积进行格式化，但我将停止简化，并介绍下面的公式。

给定：

$$p = p_0 + p_1*i + p_2*j + p_3*k = p_0 + p_v$$
$$q = q_0 + q_1*i + q_2*j + q_3*k = q_0 + q_v$$

则：

$$r = p*q = (p_0*q_0 - (p_v \cdot q_v)) + (p_0*q_v + q_0*p_v + p_v \times q_v)$$
$$= r_0 + r_v$$

注意：运算符×表示标准向量叉积，它对四元数的虚部向量进行计算，并将虚部视为标准 3D 向量。

由于点积总是标量，而叉积总是向量，因此第一项（$p_0*q_0-(p_v \cdot q_v)$）为实部，而（$p_0*q_v+q_0*p_v+p_v \times q_v$）是向量（虚部）$r_v$。必须承认，四元数乘法在数学上非常难看！

另外，乘法恒等元（四元数数学中的“1”）为：

$$q_1 = 1 + 0*i + 0*j + 0*k$$

这是正确的，因为 $q * q_1 = q = q_1 * q$。

4. 共轭四元数

对于四元数 q，计算其共轭 q^*的方法与复数相同：只需要将虚数分量 q_v 的符号反过来即可。

给定：

$$q = q_0 + q_1*i + q_2*j + q_3*k = q_0 + q_v$$

可以通过将虚部的符号反转，来计算四元数的共轭：

$$q^* = q_0 - q_1*i - q_2*j - q_3*k = q_0 - q_v$$

来看一下 q 与 q^*的积，这相当有意思：

$$q * q^* = (q_0 + q_v) * (q_0 + (-q_v))$$
$$= q_0*q_0 - (q_v \cdot (-q_v)) + q_0*q_v + q_0*(-q_v) + (q_v \times (-q_v))$$
$$= q_0^2 + q_1^2 + q_2^2 + q_3^2 + (q_0*q_v - q_0*q_v) + (0)$$
$$= q_0^2 + q_1^2 + q_2^2 + q_3^2$$

它是各个分量系数的平方和。计算四元数的范数和倒数时，这种性质很有用。

5. 四元数的范数

四元数的范数可以按照与复数范数相同的方法来计算.

给定：

$$q = q_0 + q_1*i + q_2*j + q_3*k = q_0 + q_v$$

则：

$$|q| = sqrt(q_0^2+q_1^2+q_2^2+q_3^2) = sqrt(q * q^*)$$

当然：

$$|q|^2 = (q_0^2 + q_1^2 + q_2^2 + q_3^2) = (q^* q^*)$$

数学知识：$(q * q^*)=(q^* * q)$. 四元数与其共轭的积与次序无关，但一般来说，$q*p \neq p*q$。另外，$q+q^*=2*q_0$。

6. 倒数

四元数的倒数对我们来说特别重要，因为它可以简化四元数旋转。实际上，之前介绍的所有内容都是为了使用四元数来旋转向量，而倒数又是这种操作所必不可少的。下面来介绍倒数。

给定四元数 q，我们要找到另一个四元数 q^{-1}，使得下列等式成立：

$$q*q^{-1} = 1 = q^{-1}*q$$

将等式两边同时乘以共轭四元数 q^*：

$(q*q^{-1})*q^* = 1 = (q^{-1}*q)*q^*=q^*$

由于括号中的积为 1，而 1 乘以共轭和共轭乘以 1 的结果都是共轭，所以没有改变任何东西，也没有创建一个不成立的等式。下面通过这个等式来计算倒数，单独看一下等式的右边：

$(q^{-1} * q) * q^* = q^*$

或

$q^{-1}* (q * q^*) = q^*$

由于：

$(q * q^*) = |q|^2$

因此倒数为：

$q^{-1} = q^*/|q|^2$

如果 q 是一个单位四元数，则$|q|^2 = 1$，可以进一步将倒数简化为：

$q^{-1} = q^*$

上面的公式是使用四元数执行旋转成为可能的全部原因。因此，大多数时候，我们都假设所有四元数是单位四元数，这样便可以使用上面的公式，而不会出现任何问题。

4.10.3　四元数的应用

学习四元数的知识后，读者可能认为它们是一些没有什么实际用途的数学对象。然而，四元数对于 3D 图形学中的几种功能是非常有用的：旋转和旋转插值，如图 4.44 所示。图有中两个相机方向：$camera_1 = (\alpha_1, \phi_1, \theta_1)$和 $camera_2 = (\alpha_2, \phi_2, \theta_2)$，它们分别是由一组相对于 x、y、z 轴旋转角度定义的。如何将相机指向这些方向并不重要，重要的是，如何从 $camera_1$ 平滑过渡到 $camera_2$ 方向呢？当然，可以根据角度进行线性插值，但这样做将出现很多问题，如抖动或当相机正好与坐标轴平行时减少自由度。

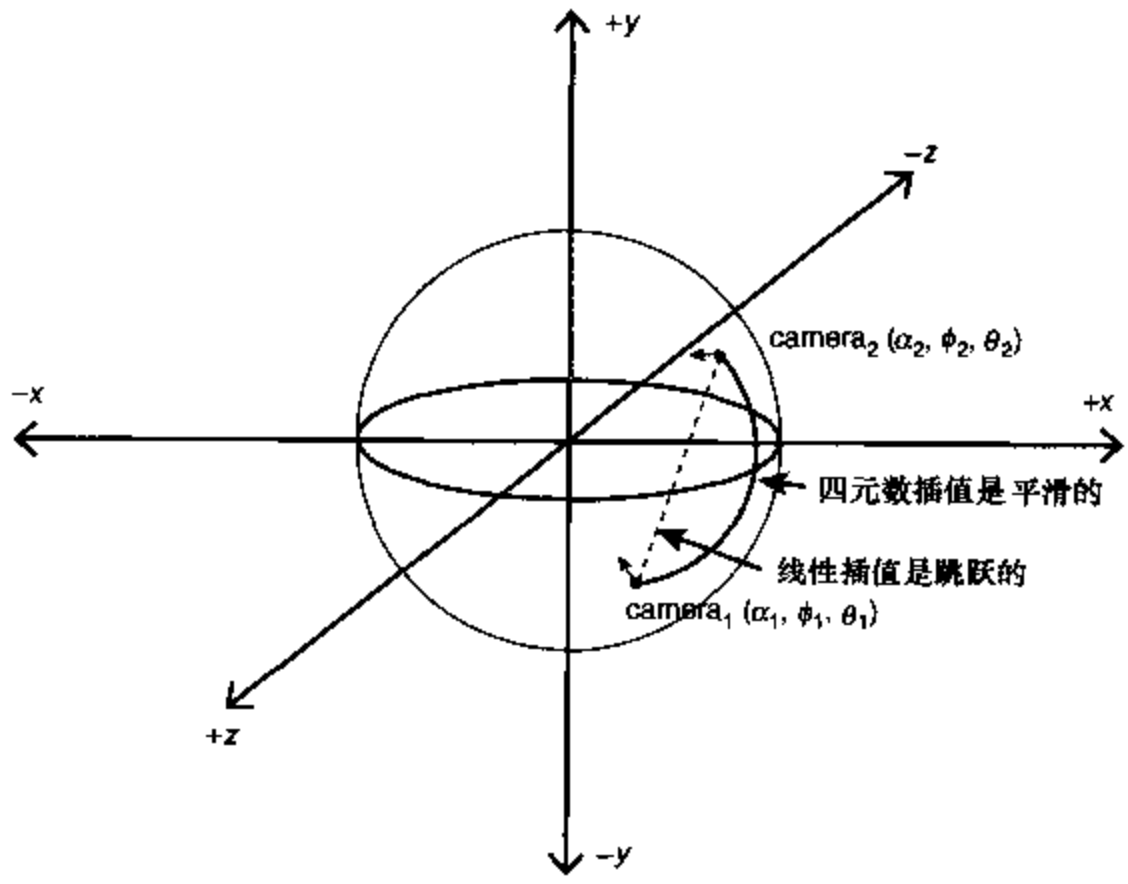


图 4.44　在两个朝向之间进行插值

四元数由于其 4D 属性，在处理这些问题方面，比标准角度和旋转矩阵更好；速度可能慢些，但值得使用。我们不会使用四元数执行标准 3D 旋转和操作，但将使用它们进行高级相机操作，这比使用角度操作相机要直观得多。下面将介绍如何使用四元数执行向量旋转；以后讨论相机时，将介绍如何使用四元数进行相机方向插值等。

四元数旋转

您可能会问，四元数变换表示什么运算呢？换句话说，它们有用途吗？答案是肯定的。其中最有用的操作之一是旋转向量。我很乐意介绍所有的数学过程，最后得到结果，但是由于时间和篇幅的限制，只能简略地介绍数学内容，以使您不会错过任何内容。最终结果是，给定向量 $\mathbf{v}=<x, y, z>$，其四元数形式 $\mathbf{v}_q=<0, x, y, z>$（这里使用一个虚构的 $0=q0$，以便将 v 表示为四元数），以及一个单位四元数 **q**，旋转 $\mathbf{v}_q$ 的运算如下：

右手坐标系：

a.$\mathbf{v}'_q=\mathbf{q}^*\mathbf{v}_q\mathbf{q}$，顺时针旋转

b.$\mathbf{v}'_q=\mathbf{q}\mathbf{v}_q\mathbf{q}^*$，逆时针旋转

左手坐标系：

c.$\mathbf{v}'_q=\mathbf{q}\mathbf{v}_q\mathbf{q}^*$，顺时针旋转

d.$\mathbf{v}'_q=\mathbf{q}^*\mathbf{v}_q\mathbf{q}$，逆时针旋转

数学知识：在上述运算中，只有单位四元数才能使用 **q***；如果不是单位四元数，应该使用 **q** 的倒数。

$\mathbf{v}_q$ 是一个用 $q_0=0$ 的四元数表示的向量，**q** 为一个四元数，但 **q** 到底代表什么呢？它与 *x*、*y*、*z* 轴以及 $\mathbf{v}_q$ 的关系是什么呢？**q** 定义了旋转轴和要绕该轴旋转的角度。当然，$\mathbf{v}_q'$在数学上是一个 4D 向量或四元数。然而，其第一个分量 q_0 总是为零，因此可以去掉该分量，仅考虑后三个元素，将它们视为 3D 空间中的向量，代表向量 v 旋转后得到的向量。

如果您曾经绕任意轴旋转过 3D 物体，则将知道那并不容易，但现在变得非常容易。四元数 **q** 定义的旋转轴和旋转角度并不明显，但也不是太糟。从图 4.45 可知，对于给定单位四元数 $\mathbf{q}=q_0+\mathbf{q}_v$，旋转轴是向量部分 $\mathbf{q}_v$ 定义的直线，而旋转角度可以根据 q_0，使用如下变换得到：

$\mathbf{q}=\cos(\theta/2)+\sin(\theta/2)*\mathbf{v}_q$

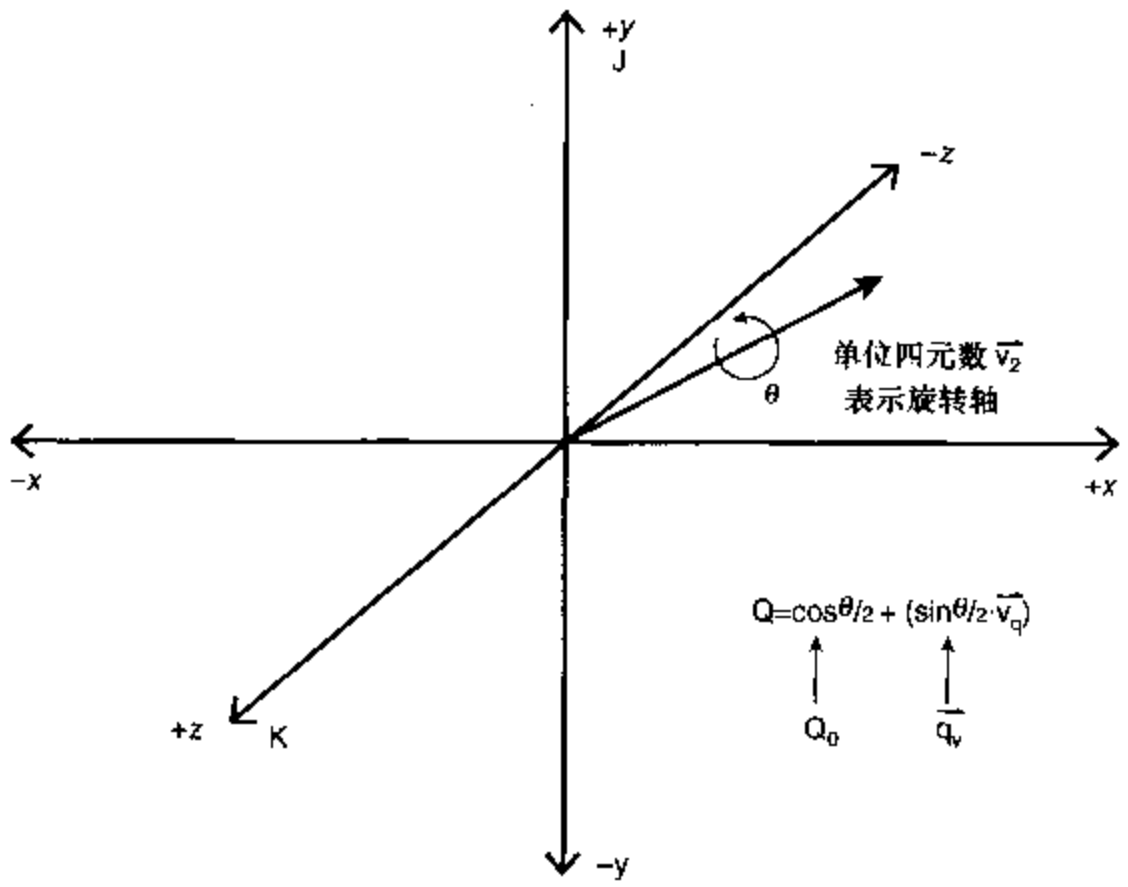


图 4.45　旋转四元数的几何解释

因此：

$q_0=\cos(\theta/2)$ and $\mathbf{q}_v=\sin(\theta/2)*\mathbf{v}_q$

注意：当然，$\mathbf{q}_v$必须为单位向量，这样 **q** 将为单位四元数。

如果不知道要绕其旋转的向量，而只知道标准欧拉旋转角，该如何办呢？可以根据角度计算下述任何一个乘积，来生成一个四元数：

$$
\begin{aligned}
\mathbf{q}_{final} &= \mathbf{q}_{x\theta} * \mathbf{q}_{y\theta} * \mathbf{q}_{z\theta} \\
&= \mathbf{q}_{x\theta} * \mathbf{q}_{z\theta} * \mathbf{q}_{y\theta} \\
&= \mathbf{q}_{y\theta} * \mathbf{q}_{x\theta} * \mathbf{q}_{z\theta} \\
&= \mathbf{q}_{y\theta} * \mathbf{q}_{z\theta} * \mathbf{q}_{x\theta} \\
&= \mathbf{q}_{z\theta} * \mathbf{q}_{x\theta} * \mathbf{q}_{y\theta} \\
&= \mathbf{q}_{z\theta} * \mathbf{q}_{y\theta} * \mathbf{q}_{x\theta} \quad \text{<- 最常见的交换}
\end{aligned}
$$

其中 x_θ 是绕 x 轴旋转的角度（倾角），y_θ 是绕 y 轴旋转的角度（偏航角），z_θ 是绕 z 轴旋转的角度（倾侧角），如图 4.46 所示。大多数人通常在 3D 引擎中使用最后一种转换顺序，但所有变换顺序都是可行的。确定变换顺序后，便可以使用下列公式来计算四元数 $\mathbf{q}_{x\theta}$、$\mathbf{q}_{y\theta}$、$\mathbf{q}_{z\theta}$，然后执行乘法来得到 $\mathbf{q}_{final}$。

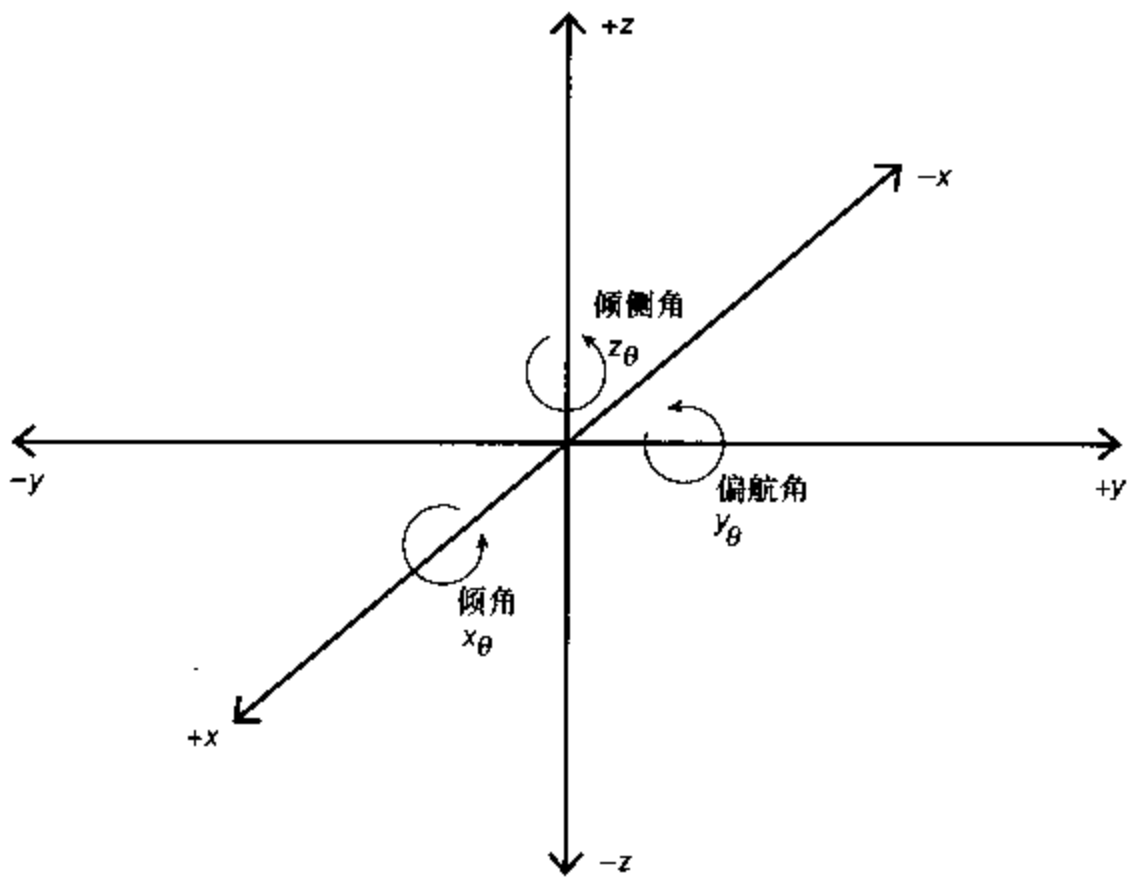


图 4.46　偏航角（yaw）、倾角（pitch）和倾侧角（Roll）

$$
\begin{aligned}
\mathbf{q}_{x\theta} &= \cos(x_\theta/2) + \sin(x_\theta/2)*\mathbf{i} + 0*\mathbf{j} + 0*\mathbf{k} \\
\mathbf{q}_\theta &= \cos(x_\theta/2),\ \mathbf{q}_v = \langle \sin(x_\theta/2),\ 0,\ 0 \rangle \\
\\
\mathbf{q}_{y\theta} &= \cos(y_\theta/2) + 0*\mathbf{i} + \sin(y_\theta/2)*\mathbf{j} + 0*\mathbf{k} \\
\mathbf{q}_\theta &= \cos(y_\theta/2),\ \mathbf{q}_v = \langle 0, \sin(y_\theta/2),\ 0 \rangle \\
\\
\mathbf{q}_{z\theta} &= \cos(z_\theta/2) + 0*\mathbf{i} + 0*\mathbf{j} + \sin(z_\theta/2)*\mathbf{k} \\
\mathbf{q}_\theta &= \cos(z_\theta/2),\ \mathbf{q}_v = \langle 0, 0, \sin(z_\theta/2) \rangle
\end{aligned}
$$

因此使用欧拉角度来旋转向量 v 时，将执行下列旋转变换：

$$\mathbf{v}_q'(0,x',y',z') = (\mathbf{q}_{z\theta} * \mathbf{q}_{y\theta} * \mathbf{q}_{x\theta}) * \mathbf{v}_q * (\mathbf{q}_{z\theta} * \mathbf{q}_{y\theta} * \mathbf{q}_{x\theta})^*$$

其中 $\mathbf{v}_q = \langle 0,\ x,\ y,\ z \rangle$，即要使用四元数方式对其进行旋转的向量或点。

当然，只需要对括号中的乘积计算一次。由于每个系数 $\mathbf{q}_{x\theta}$、$\mathbf{q}_{y\theta}$、$\mathbf{q}_{z\theta}$ 都有如下形式：

$$\mathbf{q}_{(i,j,k)} = \cos(\theta/2) + \sin(\theta/2)*(\mathbf{i},\mathbf{j},\mathbf{k})$$

因此乘积中的很多项会被消除，最终乘积 $\mathbf{q}_{x\theta} * \mathbf{q}_{y\theta} * \mathbf{q}_{z\theta}$ 可能像这样：

$$
\begin{aligned}
\mathbf{q}_0 &= \cos(_{z\theta/2})*\cos(_{y\theta/2})*\cos(_{x\theta/2}) + \sin(_{z\theta/2})*\sin(_{y\theta/2})*\sin(_{x\theta/2}) \\
\mathbf{q}_1 &= \cos(_{z\theta/2})*\cos(_{y\theta/2})*\sin(_{x\theta/2}) - \sin(_{z\theta/2})*\sin(_{y\theta/2})*\cos(_{x0/2}) \\
\mathbf{q}_2 &= \cos(_{z0/2})*\sin(_{y\theta/2})*\cos(_{x\theta/2}) + \sin(_{z0/2})*\cos(_{y\theta/2})*\sin(_{x\theta/2}) \\
\mathbf{q}_3 &= \sin(_{z\theta/2})*\cos(_{y0/2})*\cos(_{x\theta/2}) - \cos(_{z0/2})*\sin(_{y0/2})*\sin(_{x\theta/2})
\end{aligned}
$$

注意：乘积中的每个四元数都可视为一次单独的旋转操作，但是四元数乘法的计算量比矩阵乘法少得

多。因此，如果您发现自己在循环中执行了很多次矩阵乘法，可考虑将它转换为四元数进行优化——以四元数的形式执行所有矩阵运算，然后将结果转换为矩阵形式。

矩阵到四元数的转换以及四元数到矩阵的转换都非常复杂，这里没有时间进行推导。后面讨论相机时将介绍它们，以便您在合适的时候使用它们。另外，后面讨论相机运动时，将介绍四元数插值。

4.11 总　　结

本章介绍了很多数学知识：向量、矩阵、参数化方程、复数、四元数等；这只是当今编写视频游戏所需数学知识的冰山一角。

强烈建议您参考线性代数、微积分和物理学的书籍，它们会很有帮助。毫无疑问，在 3~5 年内，计算机游戏将运行真实的物理模型，所有这些都意味着数学、数学、数学！从现在开始，将不再详细介绍数学原理，而假设您已经掌握了相关的数学知识。

第 5 章　建立数学引擎

我本打算将介绍数学引擎的本章和前一章合并为一章，但由于需要介绍的内容太多，篇幅过长，所以将数学引擎单列一章。基本上，第 4 章介绍的一切内容都将在本章以某种方式在代码中实现：向量、矩阵、四元数等。本章的大部分内容都是代码，因为数学原理已经在前一章介绍过。本章不打算介绍所有函数的细节，而只选择一部分有代表性的函数。在大多数情况下，数学函数只是前一章介绍的数学知识的实现，将它们转换为了 C/C++代码。本章介绍下列内容：

- 数学引擎概述；
- 数学常量；
- 数据结构；
- 宏和内联函数；
- 函数原型；
- 全局变量；
- 坐标系支持；
- 向量支持；
- 矩阵支持；
- 2D 和 3D 参数化直线支持；
- 3D 平面支持；
- 四元数支持；
- 定点数支持；
- 浮点数支持。

5.1　数学引擎概述

如前面所述，我本打算将数学知识和实现代码安排到一章中，但由于这些内容太复杂，我又不想过度删减，因此本章继续前一章的内容，在一个数学引擎中实现前一章介绍的所有数学理论。我们的 3D 引擎将以该数学引擎为基础，根据需要添加功能，对其进行修改或改进。它不是最高效的数学库，因为没有在底层做过多的优化。

当然，在可能的情况下，我将在高层对算法进行优化。例如，计算矩阵的行列式时，我将使用共用因子扩展法（co-factor expansion），而不是直接根据公式进行计算，这样可以减少乘法运算。然而，在真正的 3D 引擎中，如果知道引擎在运行阶段需要做什么，将可以采用很多快捷方法来减少计算量。关键之处在于，我们现在有一些工具可用，可以在以后再考虑优化问题。最后，现在 1GHz 以上的 CPU 非常普遍，所以也不用太担心速度的问题。

5.1.1　数学引擎的文件结构

就文件数量而言，这个数学引擎还算比较简单。它只有两个新文件：

- T3DLIB4.CPP——C++源代码文件；
- T3DLIB4.H——头文件。

图 5.1 描述了数学引擎与游戏系统中其他文件的关系。数学库 T3DLIB4.CPP 依赖于 T3DLIB1.CPP 中的某些数据结构，因此总是需要链接 T3DLIB1.CPP|H。而 T3DLIB1.CPP|H 又依赖于 DDRAW.H，因此必须在 T3DLIB4.CPP 中包含 DDRAW.H。这意味着要在其他地方独立使用该数学库模块，需要将 T3DLIB1.CPP 中的几个函数和 T3DLIB1.H 中的#defines 语句分别复制到 T3DLIB4.CPP 和 T3DLIB4.H 中。

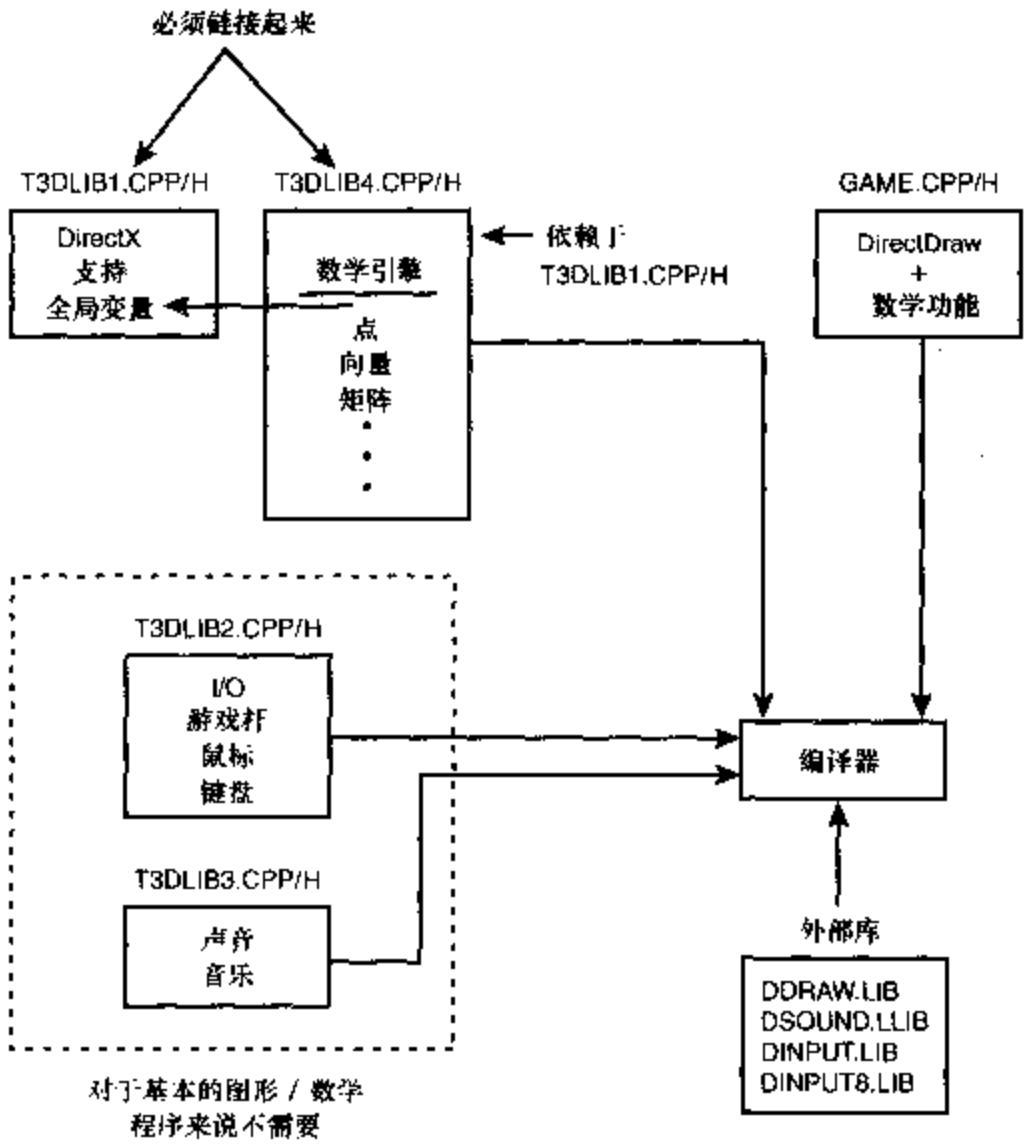


图 5.1 数学引擎与游戏系统的关系

数学引擎本身由很多函数组成（对它们进行测试并不容易），它们用于处理点、向量、直线、矩阵、四元数等。当然，该数学引擎不可能面面俱到，但对本书而言足够了。

5.1.2 命名规则

与《Windows 游戏编程大师技巧》相同，本书对函数进行升级时，如果函数只是以另一种方式（更好的方式）执行相同的操作，将在函数名称中加上一个数字后缀。例如，函数 Multiply() 在本书后面可能升级为 Multiply2()。命名规则对于每个人来说都是问题，因为您希望函数有合理的名称，但又不想输入太多的字符。在大多数情况下，我使用类/结构名作为函数名称的一部分，加上函数功能的英文描述，并使用下划线来连接。例如，将向量与矩阵相乘的函数可能这样：

```
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va,
                          MATRIX3X3_PTR mb,
                          VECTOR3D_PTR vprod);
```

上述函数声明非常明确的说明了函数的功能。OpenGL 采用类似的命名规则。上述函数名称比下面的函数名称好得多：

```
MMV3D33(...)
```

虽然在实际的引擎中，您可能使用更简单的函数命名规则，但对于本书而言，多输入几个字符将使得

事情更加清楚。

注意：另外，类名和结构名全部大写，宏名也如此。但有些内联宏（快捷函数）采用大小写混合，与标准函数类似。

5.1.3　错误处理

错误处理非常简单——根本就没有！本书是为中级/高级程序员编写的：如果您想进行错误处理和引发异常，可自行添加。大多数情况下，每个函数都将检查一些明显的错误，如被零除，但大部分函数都没有返回值，即使有，也通常是 1（TRUE，表示成功）或者 0（FALSE，表示出现了问题）。

5.1.4　关于 C++的最后说明

基于多年来开发游戏和编写游戏书籍的经验，我得出的结论是，在教学方面 C++是一种极其糟糕的语言！这种语言在生产中功能强大，我也使用它来编写游戏，但在教学方面是非常糟糕的。这里不详细说明原因，但事实确是如此。例如，在很多有关编程和数据结构的大学课程中，使用的是 Pascal、Modula II 和 Ada，为什么呢？因为它们都是结构化程度非常高的语言，学生清楚地知道每行代码的功能。在不知道类定义的情况下，很难理解 C++代码。学习算法时，何必让类、构造函数和重载运算符将问题复杂化呢？

当我兴奋地使用复杂的C++编写一个库后，最后却发现它处于C+状态(C加上函数重载和一点点C++)。之所以不使用 C++，是因为 C 语言版本与 C++版本的速度一样快，甚至更快，但更易于理解。另一方面，您可以将数学库或其他内容转换为 C++ ——如果您认为这样做有帮助的话。

我个人认为，使用重载运算符来执行矩阵和向量运算再好不过了，但从教学角度看，它很难理解。另外，每当需要更新基于 C++类的数学库时，必须编写一个新类，并派生和重载前面的类——工作量太大了。我更愿意在有函数 Multiply() 的情况下，将新函数命名为 Multiply2()。

5.2　数据结构和类型

下面介绍数学引擎使用的数据结构和类型。在很多程度上说，90%的类型和数据结构都是全新的，但也使用了 T3DLIB1.CPP|H 中的一些类型和数据结构，用于执行简单的矩阵运算，

该数学引擎支持很多数据类型，包括点、向量、矩阵、四元数、参数化直线、3D 平面、极坐标、柱面坐标、球面坐标和定点数等。下面依次介绍这些数据结构。

5.2.1　向量和点

数学引擎支持 2D、3D、4D 点和向量，其中 4D 表示格式为(x，y，z，w)的齐次坐标。然而，大多数情况下，我们不会使用 w（它总是为 1）。由于点和向量实际上是同一回事（至少，从数据上说是如此），因此使用相同的数据结构来存储它们。

另外，我决定在结构中使用共用体，以支持多种命名规则。例如，有时候使用 p.x、p.y、p.z 来表示 3D 点 p 比较合适，但有时候需要使用以数组方式（p.M[0]、p.M[1]、p.M[2]）访问 x、y、z 的算法。编写所有数据结构时，都遵循了这种方法。这样做旨在支持更多的数据访问方法，以便能够编写清晰的算法。下面是一些用于表示向量和点的数据类型：

```
// 不包含 w 分量 2D 向量和点////////////////////////
typedef struct VECTOR2D_TYP
{
union
  {
  float M[2]; // 数组存储方式

  // 独立变量存储方式
  struct
```

```
    {
    float x,y;
    }; // end struct

  }; // end union

} VECTOR2D, POINT2D, *VECTOR2D_PTR, *POINT2D_PTR;

// 不包含 w 分量的 3D 向量和点//////////////////////////
typedef struct VECTOR3D_TYP
{
union
  {
  float M[3]; // 数组存储方式

  // 独立变量存储方式
  struct
    {
    float x,y,z;
    }; // end struct

  }; // end union

} VECTOR3D, POINT3D, *VECTOR3D_PTR, *POINT3D_PTR;

// 包含 W 分量的 4D 齐次向量和点//////////////////////
typedef struct VECTOR4D_TYP
{
union
  {
  float M[4]; // 数组存储方式

  // 独立变量存储方式
  struct
    {
    float x,y,z,w;
    }; // end struct
  }; // end union

} VECTOR4D, POINT4D, *VECTOR4D_PTR, *POINT4D_PTR;
```

下面是 T3DLIB1.H 中定义的顶点数据结构：

```
// 2D 顶点
typedef struct VERTEX2DI_TYP
    {
    int x,y; // 顶点坐标
    } VERTEX2DI, *VERTEX2DI_PTR;

// 2D 顶点
typedef struct VERTEX2DF_TYP
    {
    float x,y; // the vertex
    } VERTEX2DF, *VERTEX2DF_PTR;
```

下一章处理表示 3D 物体的数据结构时，将创建表示 3D 顶点的数据结构。

5.2.2 参数化直线

大多数情况下，使用直线的参数化表示的算法通常根据描述问题的点和变量，动态地创建参数化直线，但我认为也需要一套处理显式参数化直线的函数。因此，这个数学库提供了对 2D 和 3D 参数化直线的支持。下面是 2D 参数化直线的数据结构：

```
// 2D 参数化直线 ////////////////////////////////////////
typedef struct PARMLINE2D_TYP
{
POINT2D p0; // 参数化直线的起点
```

```
POINT2D p1; // 参数化直线的终点
VECTOR2D v; // 线段的方向向量
        // |v|=|p0->p1|
} PARMLINE2D, *PARMLINE2D_PTR;
```

如图 5.2 所示，**p0** 代表起点，**p1** 代表直线的终点，而 **v** 为向量 **p0->p1**。3D 参数化直线的表示也非常容易：

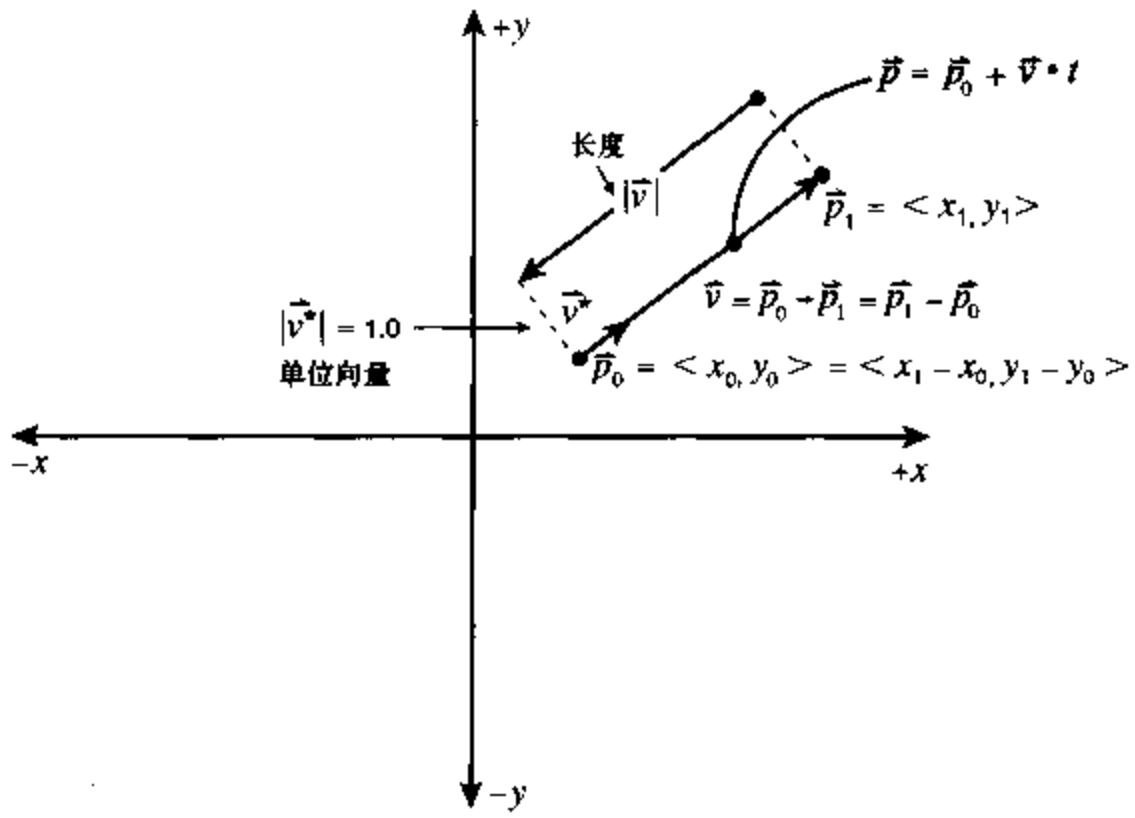


图 5.2　2D 参数化直线模型

```
// 3D 参数化直线/////////////////////////////////////////////
typedef struct PARMLINE3D_TYP
{
POINT3D p0; // 参数化直线的起点
POINT3D p1; // 参数化直线的终点
VECTOR3D v; // 线段的方向向量
        // |v|=|p0->p1|
} PARMLINE3D, *PARMLINE3D_PTR;
```

图 5.3 是一条 3D 参数化直线。**p0**、**p1** 和 **v** 的含义与 2D 参数化直线中相同。然而，图中使用的是左手坐标系。这有关系吗？没有！直线就是直线，而不管在哪种坐标系中。只有在相同的坐标系中定义和使用直线，就不会有问题。

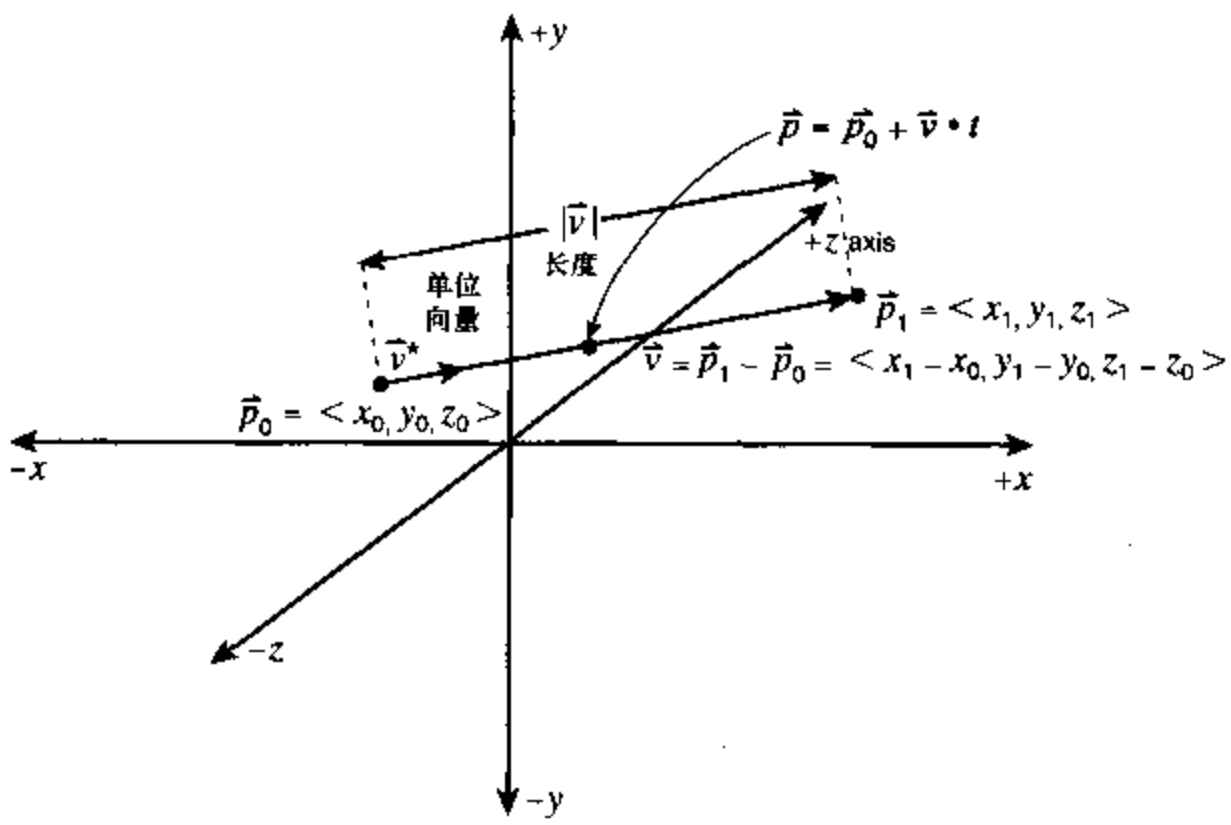


图 5.3　3D 参数化直线模型

5.2.3 3D 平面

虽然在大多数时候，需要考虑 3D 平面，但通常是在多边形语境下考虑。由于以下几种原因，创建一个 3D 平面类型是个不错的主意：以便能够执行空间算法，简化裁剪算法的编写，有助于编写 3D 直线和 3D 平面间的碰撞检测算法。表示 3D 平面的方法有很多，但归根结底它们表示的都是相同的东西。我将使用“点-法线”形式（而不是显式形式）来表示 3D 平面。然而，这并不意味着不能以下述形式存储平面：

```
a*x+b*y+c*z+d = 0
```

而只是意味着需要编写一个函数，将它转换为点-法线形式，以便存储。另外，对于很多计算来说，点-法线形式更适合，不需要进行形式转换。下面是我们将使用的数据结构：

```
// 3D 平面 ////////////////////////////////////////////////////////
typedef struct PLANE3D_TYP
{
POINT3D p0; // 平面上的点
VECTOR3D n; // 平面的法线（不必是单位向量）
} PLANE3D, *PLANE3D_PTR;
```

图 5.4 是一个 3D 平面（使用的是左手坐标系中），**p0** 是平面上的一个点，向量 n 与平面垂直的。注意，向量 n 不必是单位向量。

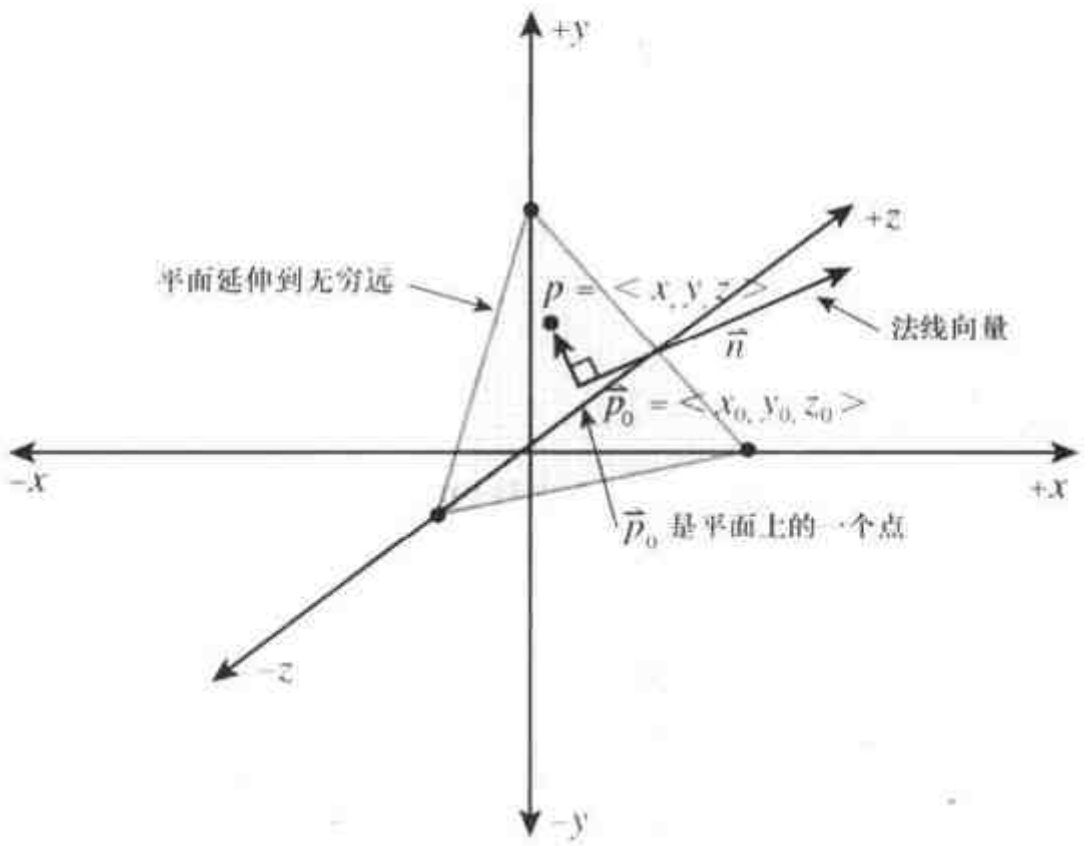


图 5.4 3D 平面模型

5.2.4 矩阵

在数学引擎中，最大的一组数据结构是矩阵。它支持 1×2、2×2、1×3、3×2、3×3、1×4、4×4 和 4×3 矩阵。1×n 矩阵实质上是一个包含 n 个分量的向量。例如，1×3 矩阵的数据结构与 VERTOR3D 相同，我们可以通过强制数据类型转换，使得为一种数据类型编写的函数可用于另一种数据类型。下面列出所有的矩阵类型。注意，很多 2×2 和 3×3 矩阵都是在 T3DLIB1.H 中定义的，所有矩阵都以先行后列的形式表示，即 [0...n][0...n] = [行_索引][列_索引]。

```
// 矩阵数据结构

// 4x4 矩阵 ///////////////////////////////////////////////////////
typedef struct MATRIX4X4_TYP
{
```

```
union
  {
  float M[4][4]; // 数组存储方式

  // 按先行后列的顺序以独立变量的方式存储
  struct
     {
     float M00, M01, M02, M03;
     float M10, M11, M12, M13;
     float M20, M21, M22, M23;
     float M30, M31, M32, M33;
     }; // end explicit names

  }; // end union

} MATRIX4X4, *MATRIX4X4_PTR;

// 4x3 矩阵 ////////////////////////////////////////////////
typedef struct MATRIX4X3_TYP
{
union
  {
  float M[4][3]; // 以数组方式存储

  // 按先行后列的顺序以独立变量的方式存储
  struct
     {
     float M00, M01, M02;
     float M10, M11, M12;
     float M20, M21, M22;
     float M30, M31, M32;
     }; // end explicit names

  }; // end union

} MATRIX4X3, *MATRIX4X3_PTR;

// 1x4 矩阵 ////////////////////////////////////////////////
typedef struct MATRIX1X4_TYP
{
union
  {
  float M[4]; // 以数组方式存储

  //按先行后列的顺序以独立变量的方式存储
  struct
     {
     float M00, M01, M02, M03;
     }; // end explicit names

   }; // end union
} MATRIX1X4, *MATRIX1X4_PTR;


// 3x3 矩阵 ////////////////////////////////////////////////
// 来自 T3DLIB1.H
typedef struct MATRIX3X3_TYP
    {
    union
    {
    float M[3][3]; // 以数组方式存储

    // 按先行后列的顺序以独立变量的方式存储
    struct
       {
       float M00, M01, M02;
       float M10, M11, M12;
```

```
        float M20, M21, M22;
        }; // end explicit names
    }; // end union
    } MATRIX3X3, *MATRIX3X3_PTR;

// 1x3矩阵 ////////////////////////////////////////////////
// 来自 T3DLIB1.H
typedef struct MATRIX1X3_TYP
    {
    union
    {
    float M[3]; // 以数组方式存储

    // 按先行后列的顺序以独立变量的方式存储
    struct
       {
       float M00, M01, M02;

       }; // end explicit names
    }; // end union
    } MATRIX1X3, *MATRIX1X3_PTR;

// 3x2矩阵 ////////////////////////////////////////////////
// 来自 T3DLIB1.H
typedef struct MATRIX3X2_TYP
    {
    union
    {
    float M[3][2]; // 以数组方式存储

    // 按先行后列的顺序以独立变量的方式存储
    struct
       {
       float M00, M01;
       float M10, M11;
       float M20, M21;
       }; // end explicit names

    }; // end union
    } MATRIX3X2, *MATRIX3X2_PTR;


// 2x2矩阵 ////////////////////////////////////////////////
typedef struct MATRIX2X2_TYP
{
union
  {
  float M[2][2]; // 以数组方式存储

  //按先行后列的顺序以独立变量的方式存储
  struct
     {
     float M00, M01;
     float M10, M11;
     }; // end explicit names

   }; // end union
} MATRIX2X2, *MATRIX2X2_PTR;

// 1x2矩阵 ////////////////////////////////////////////////
// 来自 T3DLIB1.H
typedef struct MATRIX1X2_TYP
    {
    union
    {
    float M[2]; // 以数组方式存储
```

```
// 按先行后列的顺序以独立变量的方式存储
struct
   {
   float M00, M01;

   }; // end explicit names
}; // end union
} MATRIX1X2, *MATRIX1X2_PTR;
```

请注意所有矩阵的命名规则。例如，假设定义了一个这样的 3×3 矩阵：

```
MATRIX3X3 m = {0,1,2, 3,4,5, 6,7,8 };
```

该矩阵在内存中将如下：

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

它采用先行后列的顺序排列。要访问右下角的元素，并将其设置为 100.0，下面两条语句等效（由于是共用体）：

```
m.M22 = 100;

m.M[2][2] = 100;
```

5.2.5　四元数

下一种数据类型是四元数。读者可能还记得，四元数的形式如下：

```
q = q0 + q1*i + q2*j + q3*k
```

或

```
q = q0 + <q1, q2, q3>
```

或

```
q = q0 + qv
```

四元数由 4 个分量组成，实部为 q0，向量部分为虚部。另外，由于我们将在旋转和相机语境下使用四元数，所有用实数 $q0$（或 w）和向量部分$\langle x, y, z\rangle$来表示四元数有很多优点。这样可以使用其他 VECTOR3D 函数来对四元数执行操作——可能由于它们的数据结构类似。下面来看一看四元数的数据结构：

```
// 4d 四元数 ////////////////////////////////////////////////
// 通过使用共用体，提供了多种处理四元数分量的方式
typedef struct QUAT_TYP
{
union
  {
  float M[4]; // 按 w、x、y、z 的顺序以数组方式存储

  // 以向量部分和实部的格式存储
  struct
     {
     float  q0; // 实部
     VECTOR3D qv; // 虚部（xi+yj+zk）
     };
  struct
     {
     float w,x,y,z;
     };
  }; // end union

} QUAT, *QUAT_PTR;
```

该共用体支持以三种不同的方式访问四元数：以数组方式；浮点数和向量；显式名称 w、x、y 和 z，

采用哪种方式更好取决于对四元数进行处理的算法。例如，下面定义了一个四元数，然后修改其实部：

```
QUAT q = {1, 1,2,3 };

q.w  = 5; // 使用显式名称来访问
q.q0  = 5; // 使用实部名称来访问
q.M[0] = 5; // 以数组方式来访问
```

注意：有些人可能想以<x, y, z, w>的形式来表示四元数，使其数据结构与 VECTOR4D 匹配。然而，为保持一致性，我还是采用与书写格式相同的形式。

5.2.6 角坐标系支持

虽然本书大多数时候都使用笛卡尔坐标系，但有些时候需要一种更自然地表示问题的方法。例如，构建炮塔模型（如图 5.5 所示）时，使用极坐标系将最合适，因为需要相对于坐标轴的角度方向。因此，数学引擎支持所有的 2D 和 3D 角坐标系以及这些坐标系之间的变换。支持的坐标系包括 2D 极坐标系、3D 柱面坐标系和 3D 球面坐标系。所有角度都以弧度为单位，所有转换都是针对右手坐标系的，如果您使用左手坐标系，一定要小心——如果不考虑坐标系的右手性，得到的结果可能是正确结果求负。

警告：所有角度函数都假设角度以弧度为单位，而不是以度为单位！

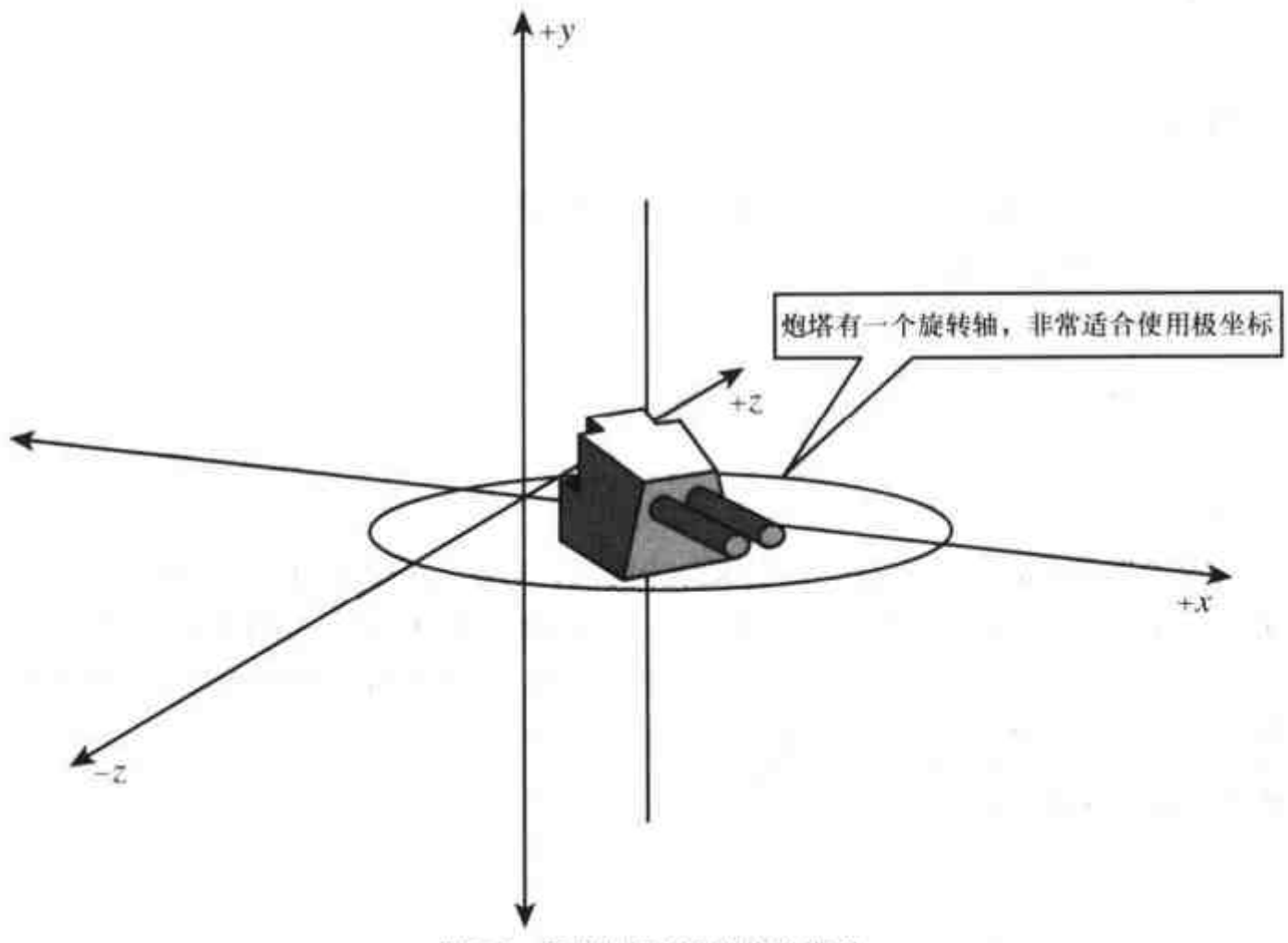


图 5.5 用极坐标描述的炮塔模型

5.2.7 2D 极坐标

2D 极坐标系如图 5.6 所示。坐标系中的点用距离（离原点或极点的距离）和方向（角度）θ表示。因此点 **p** 位于这样的射线上：相对于参考线（通常为 x 轴）的反时针角度为θ，且离原点的距离为 r。该数据类型转换为 C++后如下：

```
// 2D 极坐标 ////////////////////////////////////////////
typedef struct POLAR2D_TYP
{
float r;   // 半径
float theta; // 角度
} POLAR2D, *POLAR2D_PTR;
```

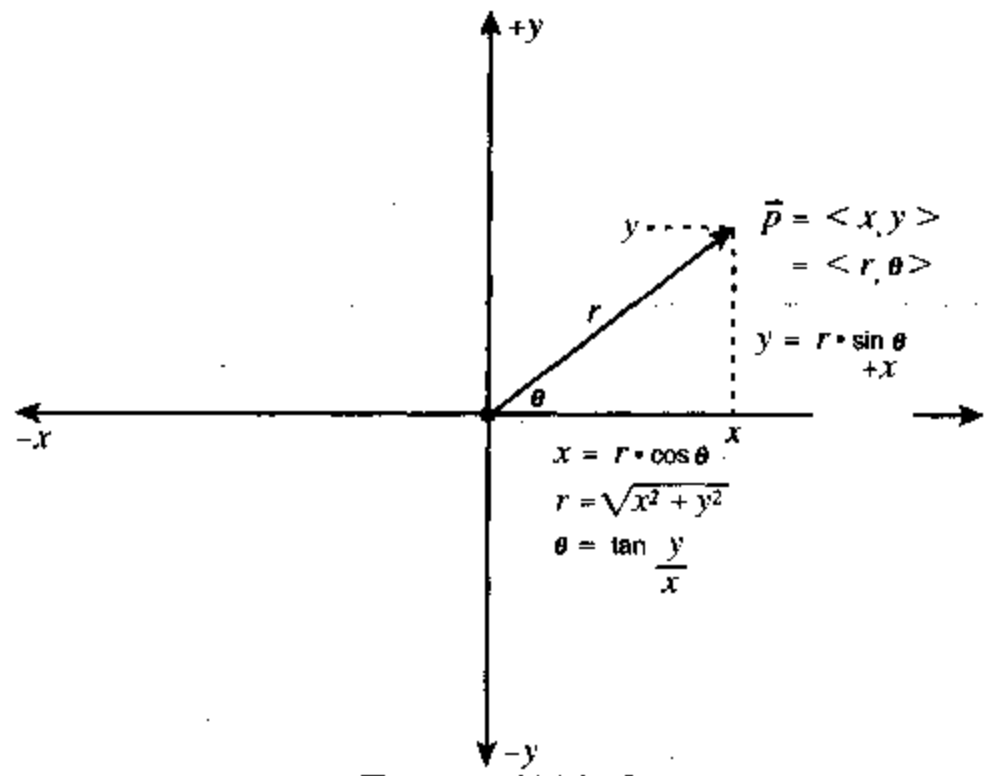


图 5.6　2D 极坐标系

5.2.8　3D 柱面坐标

3D 柱面坐标类似于 2D 极坐标系，但增加了一个 z 轴变量，如图 5.7 所示。点被定义为 $\mathbf{p}(r, \theta, z)$，表示点 **p** 离 x-y 平面的距离为 z；它在 x-y 平面上的投影位于这样的射线上：相对于参考线（通常为 x 轴）的反时针角度为 θ，且离原点的距离为 r。下面是包含这些信息的数据类型：

```
// 3D cylindrical coordinates ////////////////////////////////////
typedef struct CYLINDRICAL3D_TYP
{
float r;    // 半径
float theta; // 与 z 轴的夹角
float z;    // z 坐标
} CYLINDRICAL3D, *CYLINDRICAL3D_PTR;
```

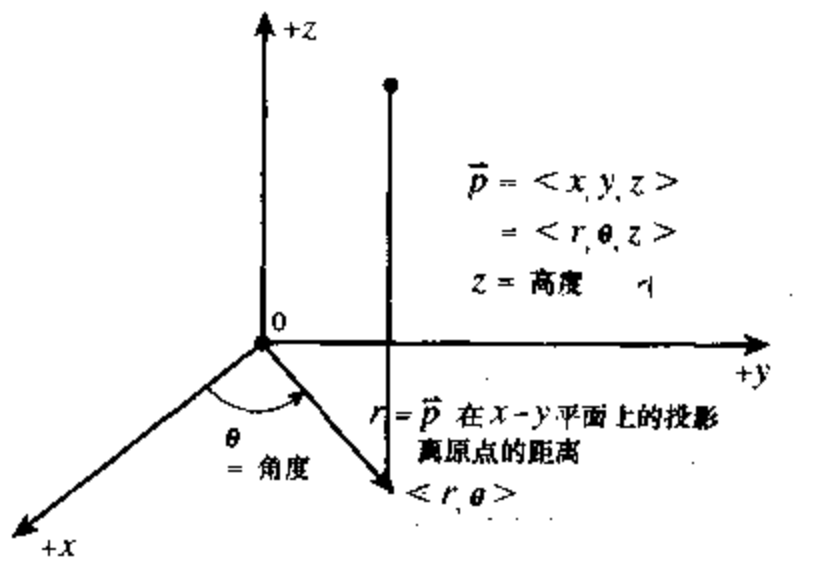


图 5.7　3D 柱面坐标系

5.2.9　3D 球面坐标

3D 球面坐标系是所有角坐标系中最复杂的。点 $\mathbf{p}(\rho, \phi, \theta)$ 由一个距离和两个角度定义，如图 5.8 所示，其中

- ρ是点 **p** 到原点 **o** 的距离。
- ϕ是原点到点 **p** 的线段与正 z 轴（这里使用的是右手坐标系）之间的夹角。
- θ 是原点 **o** 到点 **p** 的线段在 x-y 平面上的投影与正 x 轴之间的夹角，它正好是标准 2D 极角θ。另外，$0 <= \theta <= 2\pi$。

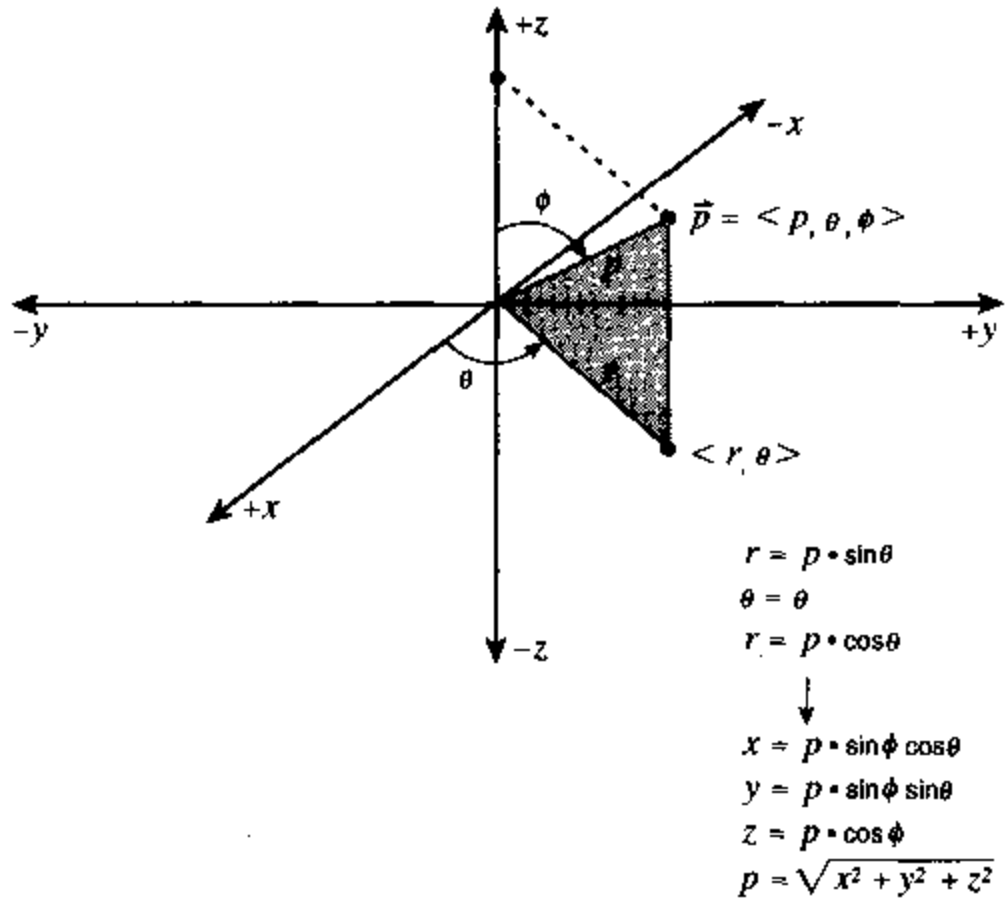


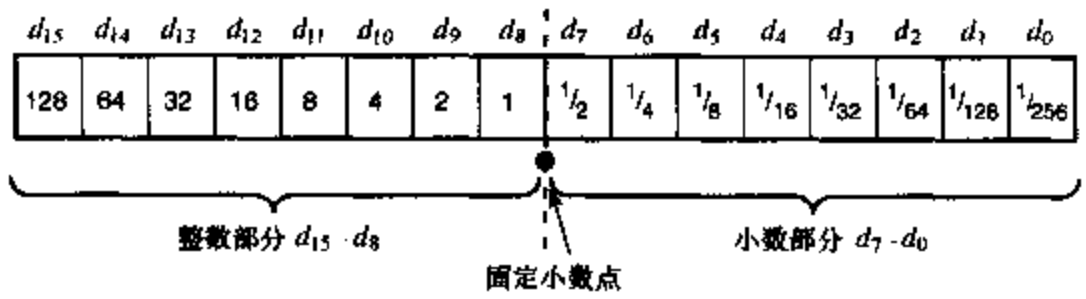
图 5.8　3D 球面坐标系

当然，这些在前一章介绍过。其数据结构为：

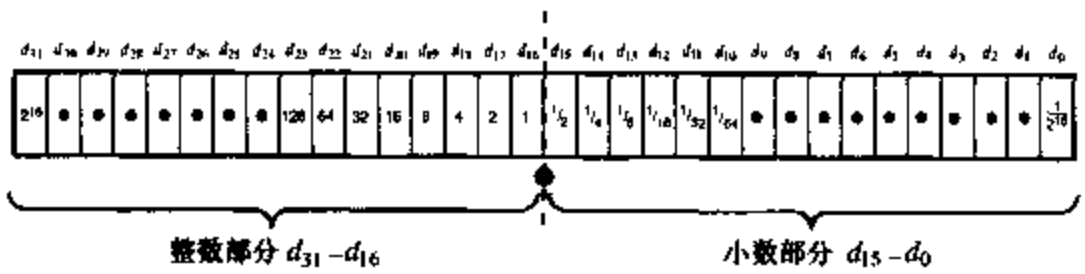
```
// 3D 球面坐标 ////////////////////////////////////
typedef struct SPHERICAL3D_TYP
{
float p;   // 到原点的距离
float theta; // 线段 o->p 和正 z 轴之间的夹角
float phi;  // 线段 o->p 在 x-y 平面上的投影与正 x 轴之间的夹角
} SPHERICAL3D, *SPHERICAL3D_PTR;
```

5.2.10　定点数

最后一种类型比较难，但的确需要它。定点数是用于以有限的精度表示浮点数的整数，以避免进行浮点处理。现在对它们的需求已不再像以前那么强烈，因为奔腾和 I64 处理器可以像定点数一样迅速地执行浮点数运算。然而，有时候仍需要使用定点数，例如无法使用的浮点的内部循环、在您只想使用整数时或在小型手持设备上时。本书的大多数数学处理都将使用 16.16 格式，如图 5.9 所示；即其中使用 16 位表示整数部分，余下 16 位表示小数部分。下面是定点数的数据类型：



A.16 位定点数格式



B.32 位定点数格式

图 5.9　定点数表示法

```
// 定点数类型 ////////////////////////////////////////////
typedef int FIXP16;
typedef int *FIXP16_PTR;
```

接下来介绍引擎使用的常量。通常，应最先介绍它们，但由于之前引用了一些数据结构，因此先介绍数据结构，后介绍它们。

5.3 数 学 常 量

数学引擎是基于一些常量的。这些常量是在 T3DLIB4.H 中定义的，但有些来自 T3DLIB1.H。下面是在 T3DLIB1.H 中定义的数学常量：

```
// T3DLIB1.H 中定义的数学常量

// 与 Pi 相关的常量
#define PI     ((float)3.141592654f)
#define PI2    ((float)6.283185307f)
#define PI_DIV_2  ((float)1.570796327f)
#define PI_DIV_4  ((float)0.785398163f)
#define PI_INV    ((float)0.318309886f)

// 与定点数运算相关的常量
#define FIXP16_SHIFT   16
#define FIXP16_MAG     65536
#define FIXP16_DP_MASK  0x0000ffff
#define FIXP16_WP_MASK  0xffff0000
#define FIXP16_ROUND_UP 0x00008000
```

注意与 PI 和定点数运算相关的常量。我们将使用定点数格式 16.16，后面介绍定点数函数时将更详细地讨论它们。下面是 T3DLIB4.H 中定义的数学常量，将它们分成了几组以便于解释：

```
// 针对于非常小的数的常量
#define EPSILON_E4 (float)(1E-4)
#define EPSILON_E5 (float)(1E-5)
#define EPSILON_E6 (float)(1E-6)
```

Epsilon 常量有助于对很小的浮点数进行数学比较。例如，执行浮点数运算时，经常需要检测一个浮点数是否等于 0.0。然而，经过几次浮点数运算后，精度将降低，很少会出现 0.0 的情况，因此一种常用策略是检测是否接近于 0.0，如下所示：

```
if (fabs(x) < 0.00001)
  {
  // 足够接近于零
  } // end if
```

下面的常量用于参数化直线函数及其返回值（后面介绍这些函数时将详细讨论）：

```
// 用于参数化直线交点的常量
#define PARM_LINE_NO_INTERSECT     0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
#define PARM_LINE_INTERSECT_EVERYWHERE 3
```

下面是一些单位矩阵，用于简化矩阵的初始化：

```
// 单位矩阵

// 4x4 单位矩阵
const MATRIX4X4 IMAT_4X4 = {1,0,0,0,
                0,1,0,0,
                0,0,1,0,
                0,0,0,1};
```

```
// 4x3 单位矩阵（从数学上说，这是不正确的）
// 本书后面将使用这种矩阵，并假设最后一列为[0 0 0 1]t
const MATRIX4X3 IMAT_4X3 = {1,0,0,
                0,1,0,
                0,0,1,
                0,0,0,};

// 3x3 单位矩阵
const MATRIX3X3 IMAT_3X3 = {1,0,0,
                0,1,0,
                0,0,1};

// 2x2 单位矩阵
const MATRIX2X2 IMAT_2X2 = {1,0,
                0,1};
```

注意：虽然只有 n×n 矩阵（即 2×2、3×3、4×4 等）才有单位矩阵，但在上面的定义中提供了一个 4×3 单位矩阵。它从数学上来说是不正确的，但在很多数学运算中，我们使用 4×3 矩阵，并假设最后一列为[0 0 0 1]t。因此，可以定义一个 4×3 单位矩阵，但在执行数学运算时需要记住它有一个额外的列。

这些就是引擎使用的全部常量。

5.4 宏和内联函数

编写数学引擎时需要考虑的一个问题是，通过函数调用对数学对象执行操作时，函数调用花费的时间可能与实际数学运算花费时间相当！因此，绝对有必要使用宏和内联函数。然而，使用大型内联函数的问题是，内联函数必须在编译时（而不是链接时）可用。也就是说，当您编译一个程序并在另一个模块中调用一个名为 func() 的函数时，编译器不需要函数 func() 的代码，而只需要其原型：

```
int func(int x, int y);
```

有了函数原型，编译器便可以在编译代码中创建函数调用，然后链接器将可以找到函数 func()，将调用解释为地址。内联函数的问题在于，其实际代码必须在编译时可用。因此，实现大型内联函数的惟一方法是，使用一个头文件和一个名为*.inc 的伪 C/C++模块，并在前者中定义内联函数，手工将后者包含在要包含的模块中。这样，编译器就可以访问内联函数的实际代码。图 5.10 说明了这种设置。

我想说的是：编写数学引擎时，所有函数都应该是内联的；对于只有几行代码的函数，运行代码花费的时间往往与设置堆栈结构、执行调用并返回结果花费的时间差不多。例如，在内部循环中调用下述函数的后果将非常严重：

```
void Plot_Pixel(UCHAR *video_buffer, int mempitch, int x, int y, int color)
{
// 绘制一个像素
video_buffer[x+y*mempitch] = color;
} // end Plot_Pixel
```

如果在程序中调用它，将生成一个函数调用。例如，请看下述代码：

```
void main()
{
UCHAR *video_buffer = (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);
for (int y=0; y < SCREEN_HEIGHT; y++)
     for (int x=0; x < SCREEN_WIDTH; x++)
        Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5); // 速度缓慢的调用

} // end main
```

虽然这个函数看起来没什么问题，来看一下 Visual C++ 6.0 编译器生成的代码：

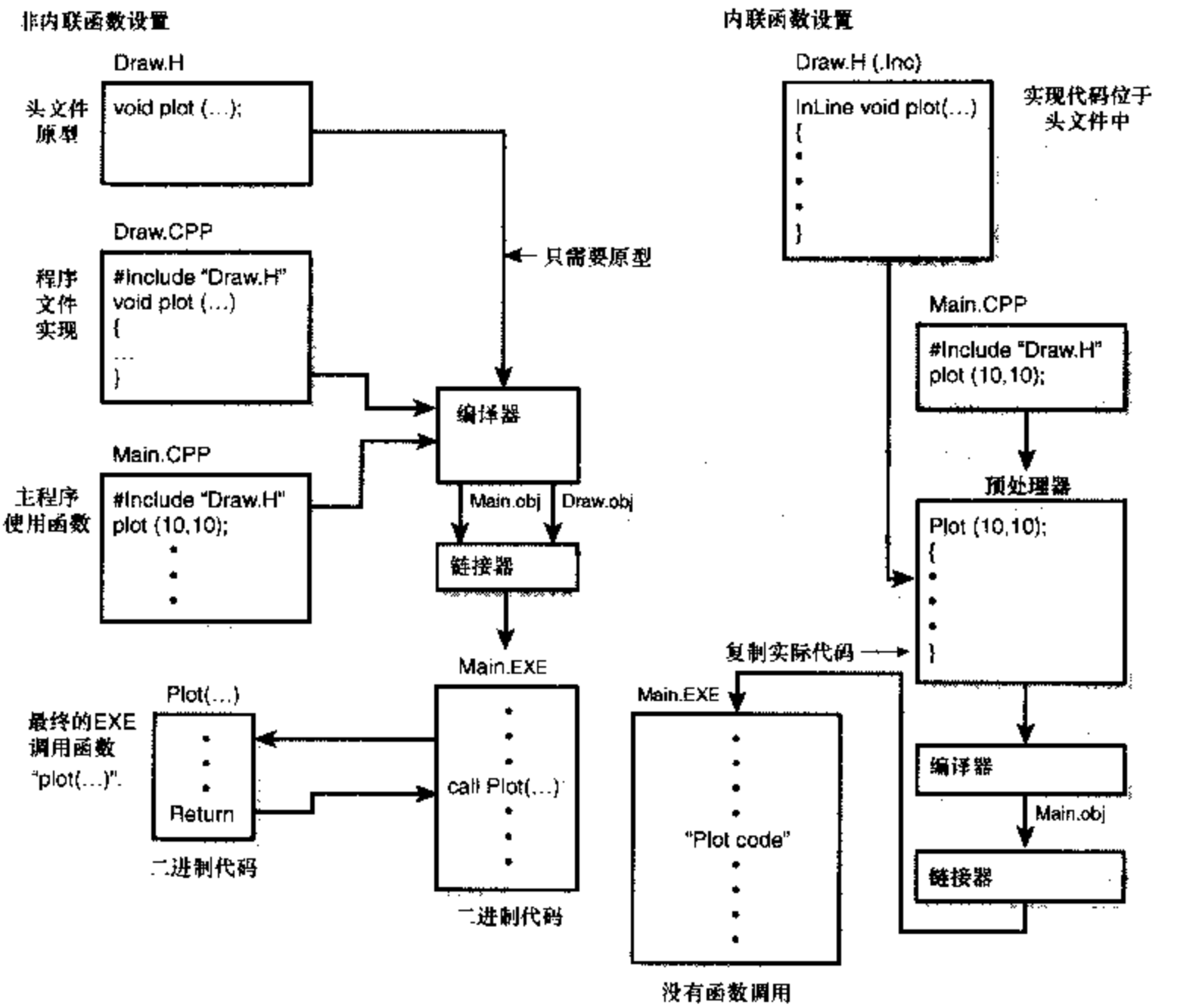


图 5.10　与编译器和链接器相关的内联过程

```
_main PROC NEAR

; 52  : {

    push  ebp
    mov  ebp, esp
    sub  esp, 12                    ; 0000000cH

; 53  : UCHAR *video_buffer = (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);

    push  307200                    ; 0004b000H
    call  _malloc
    add  esp, 4
    mov  DWORD PTR _video_buffer$[ebp], eax

; 54  :
; 55  : for (int y=0; y < SCREEN_HEIGHT; y++)

    mov  DWORD PTR _y$[ebp], 0
    jmp  SHORT $L43914
$L43915:
    mov  eax, DWORD PTR _y$[ebp]
    add  eax, 1
    mov  DWORD PTR _y$[ebp], eax
$L43914:
    cmp  DWORD PTR _y$[ebp], 480        ; 000001e0H
    jge  SHORT $L43916

; 56  :    for (int x=0; x < SCREEN_WIDTH; x++)
```

```
    mov  DWORD PTR _x$43917[ebp], 0
    jmp  SHORT $L43918
$L43919:
    mov  ecx, DWORD PTR _x$43917[ebp]
    add  ecx, 1
    mov  DWORD PTR _x$43917[ebp], ecx
$L43918:
    cmp  DWORD PTR _x$43917[ebp], 640     ; 00000280H
    jge  SHORT $L43920

; 57  :         Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5);

    push 5
    mov  edx, DWORD PTR _y$[ebp]
    push edx
    mov  eax, DWORD PTR _x$43917[ebp]
    push eax
    push 640                    ; 00000280H
    mov  ecx, DWORD PTR _video_buffer$[ebp]
    push ecx
    call ?Plot_Pixel@@YAXPAEHHHH@Z       ; Plot_Pixel
    add  esp, 20                ; 00000014H
    jmp  SHORT $L43919
$L43920:
    jmp  SHORT $L43915
$L43916:

; 58  :
; 59  : } // end main

    mov  esp, ebp
    pop  ebp
    ret  0
_main  ENDP
```

其中以粗体显示的是调用函数 Plot_Pixel()时设置堆栈结构、调用、然后返回需要的代码。而函数 Plot_Pixel()本身的代码如下：

```
?Plot_Pixel@@YAXPAEHHHH@Z PROC NEAR        ; Plot_Pixel

; 45  : {

    push ebp
    mov  ebp, esp

; 46  : // plots a pixel
; 47  : video_buffer[x+y*mempitch] = color;
    mov  eax, DWORD PTR _y$[ebp]
    imul  eax, DWORD PTR _mempitch$[ebp]
    mov  ecx, DWORD PTR _x$[ebp]
    add  ecx, eax
    mov  edx, DWORD PTR _video_buffer$[ebp]
    mov  al, BYTE PTR _color$[ebp]
    mov  BYTE PTR [edx+ecx], al

; 48  : } // end Plot_Pixel

    pop  ebp
    ret  0
?Plot_Pixel@@YAXPAEHHHH@Z ENDP             ; Plot_Pixel
```

通过比较可以知道，调用函数 Plot_Pixel()所需完成的工作与该函数本身完成的工作几乎差不多！如果使用内联函数，便可以省去函数调用的开销。下面是相同的范例，但这次我命令编译器将函数作为内联的：

```
_main  PROC NEAR
```

```
; 52  : {

    push  ebp
    mov  ebp, esp
    sub  esp, 16                    ; 00000010H

; 53  : UCHAR *video_buffer = (UCHAR *)malloc(SCREEN_WIDTH*SCREEN_HEIGHT);

    push  307200                    ; 0004b000H
    call  _malloc
    add  esp, 4
    mov  DWORD PTR _video_buffer$[ebp], eax

; 54  :
; 55  : for (int y=0; y < SCREEN_HEIGHT; y++)

    mov  DWORD PTR _y$[ebp], 0
    jmp  SHORT $L43914
$L43915:
    mov  eax, DWORD PTR _y$[ebp]
    add  eax, 1
    mov  DWORD PTR _y$[ebp], eax
$L43914:
    cmp  DWORD PTR _y$[ebp], 480        ; 000001e0H
    jge  SHORT $L43916

; 56  :    for (int x=0; x < SCREEN_WIDTH; x++)

    mov  DWORD PTR _x$43917[ebp], 0
    jmp  SHORT $L43918
$L43919:
    mov  ecx, DWORD PTR _x$43917[ebp]
    add  ecx, 1
    mov  DWORD PTR _x$43917[ebp], ecx
$L43918:
    cmp  DWORD PTR _x$43917[ebp], 640     ; 00000280H
    jge  SHORT $L43920

; 57  :       Plot_Pixel(video_buffer, SCREEN_WIDTH, x,y,5);

    mov  DWORD PTR $T43941[ebp], 5
    mov  edx, DWORD PTR _y$[ebp]
    imul  edx, 640                  ; 00000280H
    mov  eax, DWORD PTR _x$43917[ebp]
    add  eax, edx
    mov  ecx, DWORD PTR _video_buffer$[ebp]
    mov  dl, BYTE PTR $T43941[ebp]
    mov  BYTE PTR [ecx+eax], dl
    jmp  SHORT $L43919
$L43920:
    jmp  SHORT $L43915
$L43916:

; 58  :
; 59  : } // end main
```

其中以粗体显示的是调用内联函数 Plot_Pixel()的代码，它要好得多！虽然代码更多了，但速度至少提高了 200%。

当函数体很小时，应将函数声明为内联的，但实现起来很麻烦，因为没有管理这些文件的好方法。我喜欢.H 和.CPP 文件，但不喜欢将所有内联函数代码都放到.H 文件中，而新增一个.INC 文件无疑是自找麻烦。我的解决方法是：尽可能地使用#define 宏，只要您认为在多次调用宏参数时不会导致问题。

另外，如果内联函数不大，将它们放到头文件中也是可以的，因为大多数程序员不会到.H 文件中查找源代码，而只到.CPP 文件中查找源代码。另外，.H 文件中包含较多的代码时，将导致头文件的编译时间加长（当然，使用预编译头文件可以解决这种问题）。请牢记，当编写好引擎后，您可能想将很多代码较少的

函数转换为内联函数。这样做的问题在于，.CPP 文件中的所有代码都将被移到.H 文件中，这可能导致源代码的难以管理。

对于这种问题，我试图找到一个平衡点：只将简单函数转换为内联宏，并将它们放到.H 文件中；而所有真正函数仍放在.CPP 文件中。然而，有时我可能将一真正的际函数转换为内联函数，而将一些内联函数转换为非内联函数并将其放到.CPP 文件中，但我想读者应该明白了我的意思。数学引擎的速度必须非常快，因此应尽可能地使用内联函数。

下面介绍数学引擎使用的全部宏。同样，根据用途将它们分成了不同的组：向量、矩阵、直线、平面、四元数等。最后，宏和内联函数还有注释，描述每组函数的功能（毕竟您很难根据 5 行代码就判断出其功能）。

5.4.1 通用宏

通用宏都是在 T3DLIB.H 中定义的，如下所示：

```
// 计算两个表达式中较大和较小的一个
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define MAX(a, b) (((a) > (b)) ? (b) : (a))

// 交换变量的值
#define SWAP(a,b,t) {t=a; a=b; b=t;}

// 数学宏
#define DEG_TO_RAD(ang) ((ang)*PI/180.0)
#define RAD_TO_DEG(rads) ((rads)*180.0/PI)

#define RAND_RANGE(x,y) ( (x) + (rand()%((y)-(x)+1)))
```

这些宏都非常简单。我发现其中的 RAND_RANGE()、MIN()和 MAX()都很有用。

5.4.2 点和向量宏

如本章前面指出的，在 2D、3D 和 4D 空间中，点和向量的数据格式相同。因此，我将所有与点和向量相关的宏分成一组。

另外，有些宏是以内联函数的方式编写的，以帮助类型检查。还有很多函数都太长，无法编写成 #defined 宏。

```
// 向量宏、4D 向量的 w 被设置为 1
// 向量归零宏
inline void VECTOR2D_ZERO(VECTOR2D_PTR v)
{(v)->x = (v)->y = 0.0;}

inline void VECTOR3D_ZERO(VECTOR3D_PTR v)
{(v)->x = (v)->y = (v)->z = 0.0;}

inline void VECTOR4D_ZERO(VECTOR4D_PTR v)
{(v)->x = (v)->y = (v)->z = 0.0; (v)->w = 1.0;}

// 使用分量初始化向量的宏
inline void VECTOR2D_INITXY(VECTOR2D_PTR v, float x, float y)
{(v)->x = (x); (v)->y = (y);}

inline void VECTOR3D_INITXYZ(VECTOR3D_PTR v, float x, float y, float z)
{(v)->x = (x); (v)->y = (y); (v)->z = (z);}

inline void VECTOR4D_INITXYZ(VECTOR4D_PTR v, float x,float y,float z)
{(v)->x = (x); (v)->y = (y); (v)->z = (z); (v)->w = 1.0;}


// 使用另一个向量来初始化向量的宏
inline void VECTOR2D_INIT(VECTOR2D_PTR vdst, VECTOR2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void VECTOR3D_INIT(VECTOR3D_PTR vdst, VECTOR3D_PTR vsrc)
```

```
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }

inline void VECTOR4D_INIT(VECTOR4D_PTR vdst, VECTOR4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }


// 复制向量的宏
inline void VECTOR2D_COPY(VECTOR2D_PTR vdst, VECTOR2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void VECTOR3D_COPY(VECTOR3D_PTR vdst, VECTOR3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }
inline void VECTOR4D_COPY(VECTOR4D_PTR vdst, VECTOR4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }


// 初始化点的宏
inline void POINT2D_INIT(POINT2D_PTR vdst, POINT2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void POINT3D_INIT(POINT3D_PTR vdst, POINT3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }

inline void POINT4D_INIT(POINT4D_PTR vdst, POINT4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }

// 复制点的宏
inline void POINT2D_COPY(POINT2D_PTR vdst, POINT2D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; }

inline void POINT3D_COPY(POINT3D_PTR vdst, POINT3D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y; (vdst)->z = (vsrc)->z; }

inline void POINT4D_COPY(POINT4D_PTR vdst, POINT4D_PTR vsrc)
{(vdst)->x = (vsrc)->x; (vdst)->y = (vsrc)->y;
(vdst)->z = (vsrc)->z; (vdst)->w = (vsrc)->w; }
```

正如您看到的，几乎有完成任何操作的宏：归零、使用各种数据类型初始化以及复制。有很多是重复的，但谁在乎呢？

5.4.3 矩阵宏

下面是矩阵宏。注意，有些是#define 宏，而另外一些是较短的内联函数。

```
// 矩阵宏

// 清空矩阵的宏
#define MAT_ZERO_2x2(m) {memset((void *)(m), 0, sizeof(MATRIX2x2));}
#define MAT_ZERO_3x3(m) {memset((void *)(m), 0, sizeof(MATRIX3x3));}
#define MAT_ZERO_4x4(m) {memset((void *)(m), 0, sizeof(MATRIX4x4));}
#define MAT_ZERO_4x3(m) {memset((void *)(m), 0, sizeof(MATRIX4x3));}
```

注意：很多编译器在调用函数 memset( )时使用逐字节填充。可以通过对矩阵使用 4 字节清零来获得更快的速度。另外，一般来说，整数与相应浮点数的表示并不相同；例如，（float）5.0 完全不同于（int）5。然而，32 位浮点数 0.0 与 32 位整数 0 的表示是相同的，因此可以利用这种巧合，使用内存函数快速地清除浮点数组。

```
// 设置单位矩阵的宏
#define MAT_IDENTITY_2x2(m) {memcpy((void *)(m), /
    (void *)&IMAT_2x2, sizeof(MATRIX2x2));}

#define MAT_IDENTITY_3x3(m) {memcpy((void *)(m), /
    (void *)&IMAT_3x3, sizeof(MATRIX3x3));}
```

```
#define MAT_IDENTITY_4×4(m) {memcpy((void *)(m), /
    (void *)&IMAT_4×4, sizeof(MATRIX4×4));}

#define MAT_IDENTITY_4×3(m) {memcpy((void *)(m), /
    (void *)&IMAT_4×3, sizeof(MATRIX4X3));}

// 复制矩阵的宏
#define MAT_COPY_2×2(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
         (void *)(src_mat), sizeof(MATRIX2×2) ); }
#define MAT_COPY_3×3(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
         (void *)(src_mat), sizeof(MATRIX3×3) ); }
#define MAT_COPY_4×4(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
         (void *)(src_mat), sizeof(MATRIX4×4) ); }
#define MAT_COPY_4×3(src_mat, dest_mat) {memcpy((void *)(dest_mat), /
         (void *)(src_mat), sizeof(MATRIX4×3) ); }

// 对矩阵进行转置的宏
inline void MAT_TRANSPOSE_3X3(MATRIX3X3_PTR m)
{ MATRIX3X3 mt;
mt.M00 = m->M00; mt.M01 = m->M10; mt.M02 = m->M20;
mt.M10 = m->M01; mt.M11 = m->M11; mt.M12 = m->M21;
mt.M20 = m->M02; mt.M21 = m->M12; mt.M22 = m->M22;
memcpy((void *)m,(void *)&mt, sizeof(MATRIX3X3)); }

inline void MAT_TRANSPOSE_4X4(MATRIX4X4_PTR m)
{ MATRIX4X4 mt;
mt.M00 = m->M00; mt.M01 = m->M10; mt.M02 = m->M20; mt.M03 = m->M30;
mt.M10 = m->M01; mt.M11 = m->M11; mt.M12 = m->M21; mt.M13 = m->M31;
mt.M20 = m->M02; mt.M21 = m->M12; mt.M22 = m->M22; mt.M23 = m->M32;
mt.M30 = m->M03; mt.M31 = m->M13; mt.M32 = m->M22; mt.M33 = m->M33;
memcpy((void *)m,(void *)&mt, sizeof(MATRIX4X4)); }

inline void MAT_TRANSPOSE_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mt)
{ mt->M00 = m->M00; mt->M01 = m->M10; mt->M02 = m->M20;
 mt->M10 = m->M01; mt->M11 = m->M11; mt->M12 = m->M21;
 mt->M20 = m->M02; mt->M21 = m->M12; mt->M22 = m->M22; }

inline void MAT_TRANSPOSE_4X4(MATRIX4X4_PTR m, MATRIX4X4_PTR mt)
{ mt->M00 = m->M00; mt->M01 = m->M10; mt->M02 = m->M20; mt->M03 = m->M30;
 mt->M10 = m->M01; mt->M11 = m->M11; mt->M12 = m->M21; mt->M13 = m->M31;
 mt->M20 = m->M02; mt->M21 = m->M12; mt->M22 = m->M22; mt->M23 = m->M32;
 mt->M30 = m->M03; mt->M31 = m->M13; mt->M32 = m->M22; mt->M33 = m->M33;}

// 小型内联函数，可以将其转换为宏

// 矩阵和向量列互换宏
inline void MAT_COLUMN_SWAP_2X2(MATRIX2X2_PTR m, int c, MATRIX1X2_PTR v)
{ m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1]; }

inline void MAT_COLUMN_SWAP_3X3(MATRIX3X3_PTR m, int c, MATRIX1X3_PTR v)
{ m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1]; m->M[2][c]=v->M[2]; }

inline void MAT_COLUMN_SWAP_4X4(MATRIX4X4_PTR m, int c, MATRIX1X4_PTR v)
{m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1];
 m->M[2][c]=v->M[2]; m->M[3][c]=v->M[3]; }

inline void MAT_COLUMN_SWAP_4X3(MATRIX4X3_PTR m, int c, MATRIX1X4_PTR v)
{m->M[0][c]=v->M[0]; m->M[1][c]=v->M[1];
 m->M[2][c]=v->M[2]; m->M[3][c]=v->M[3]; }
```

注意所有矩阵函数都使用指针。在数学库的大多数函数中都采用这一方案。有些时候使用真正的对象是适当的，但是很多时候执行栈操作和复制会消耗太多的系统资源。

5.4.4 四元数

接下来我们要介绍的是关于四元数的宏。所有四元数函数（包括 T3DLIB4.CPP 中的四元数函数）都应

该是内联宏；它们将使得代码更加复杂，但是需要记住。下面是关于四元数的宏：

```
// 四元数宏
inline void QUAT_ZERO(QUAT_PTR q)
{(q)->x = (q)->y = (q)->z = (q)->w = 0.0;}

inline void QUAT_INITWXYZ(QUAT_PTR q, float w, float x,float y,float z)
{ (q)->w = (w); (q)->x = (x); (q)->y = (y); (q)->z = (z); }

inline void QUAT_INIT_VECTOR3D(QUAT_PTR q, VECTOR3D_PTR v)
{ (q)->w = 0; (q)->x = (v->x); (q)->y = (v->y); (q)->z = (v->z); }

inline void QUAT_INIT(QUAT_PTR qdst, QUAT_PTR qsrc)
{(qdst)->w = (qsrc)->w; (qdst)->x = (qsrc)->x;
 (qdst)->y = (qsrc)->y; (qdst)->z = (qsrc)->z; }

inline void QUAT_COPY(QUAT_PTR qdst, QUAT_PTR qsrc)
{(qdst)->x = (qsrc)->x; (qdst)->y = (qsrc)->y;
(qdst)->z = (qsrc)->z; (qdst)->w = (qsrc)->w; }
```

它们大多都是根据所需的格式初始化四元数的宏。

5.4.5　定点数宏

最后介绍定点数宏，用于对定点数进行转换和提取。其他定点数函数都将编写为真正的函数。它们本来也应该编写为内联函数，但乘法和除法函数需要很多内联汇编语言，我不想将它们都放到头文件中。

```
// 从 16.16 格式的定点数中提取整数部分和小数部分
#define FIXP16_WP(fp) ((fp) >> FIXP16_SHIFT)
#define FIXP16_DP(fp) ((fp) && FIXP16_DP_MASK)

// 将整数和浮点数转换为 16.16 格式的定点数
#define INT_TO_FIXP16(i) ((i) << FIXP16_SHIFT)
#define FLOAT_TO_FIXP16(f) (((float)(f) * (float)FIXP16_MAG+0.5))

// 将定点数转换为浮点数
#define FIXP16_TO_FLOAT(fp) ( ((float)fp)/FIXP16_MAG)
```

5.5　函 数 原 型

介绍所有的函数原型。当然，您现在还不知道它们的功能，但将有大致的了解；后面介绍函数清单时，一切都将真相大白。另外，我会标出 T3DLIB1.CPP|H 中的数学函数，以方便您查找。同样，所有函数原型都按向量、矩阵等分组列出。下面是这些函数：

```
// 通用三角函数
float Fast_Sin(float theta);
float Fast_Cos(float theta);

// 距离函数(来自 T3DLIB1.CPP|H)
int Fast_Distance_2D(int x, int y);
float Fast_Distance_3D(float x, float y, float z);

// 极坐标、柱面坐标和球面坐标函数
void POLAR2D_To_POINT2D(POLAR2D_PTR polar, POINT2D_PTR rect);
void POLAR2D_To_RectXY(POLAR2D_PTR polar, float *x, float *y);
void POINT2D_To_POLAR2D(POINT2D_PTR rect, POLAR2D_PTR polar);
void POINT2D_To_PolarRTh(POINT2D_PTR rect, float *r, float *theta);
void CYLINDRICAL3D_To_POINT3D(CYLINDRICAL3D_PTR cyl, POINT3D_PTR rect);
void CYLINDRICAL3D_To_RectXYZ(CYLINDRICAL3D_PTR cyl,
               float *x, float *y, float *z);
void POINT3D_To_CYLINDRICAL3D(POINT3D_PTR rect, CYLINDRICAL3D_PTR cyl);
void POINT3D_To_CylindricalRThZ(POINT3D_PTR rect,
                float *r, float *theta, float *z);
```

```
void SPHERICAL3D_To_POINT3D(SPHERICAL3D_PTR sph, POINT3D_PTR rect);
void SPHERICAL3D_To_RectXYZ(SPHERICAL3D_PTR sph, float *x,
                 float *y, float *z);
void POINT3D_To_SPHERICAL3D(POINT3D_PTR rect, SPHERICAL3D_PTR sph);
void POINT3D_To_SphericalPThPh(POINT3D_PTR rect,
                float *p, float *theta, float *phi);

// 2D 向量函数
void VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vsum);
VECTOR2D VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vdiff);
VECTOR2D VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Scale(float k, VECTOR2D_PTR va);
void VECTOR2D_Scale(float k, VECTOR2D_PTR va, VECTOR2D_PTR vscaled);
float VECTOR2D_Dot(VECTOR2D_PTR va, VECTOR2D_PTR vb);
float VECTOR2D_Length(VECTOR2D_PTR va);
float VECTOR2D_Length_Fast(VECTOR2D_PTR va);
void VECTOR2D_Normalize(VECTOR2D_PTR va);
void VECTOR2D_Normalize(VECTOR2D_PTR va, VECTOR2D_PTR vn);
void VECTOR2D_Build(VECTOR2D_PTR init, VECTOR2D_PTR term,
                    VECTOR2D_PTR result);
float VECTOR2D_CosTh(VECTOR2D_PTR va, VECTOR2D_PTR vb);
void VECTOR2D_Print(VECTOR2D_PTR va, char *name);

// 3D 向量函数
void VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vsum);
VECTOR3D VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vdiff);
VECTOR3D VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Scale(float k, VECTOR3D_PTR va);
void VECTOR3D_Scale(float k, VECTOR3D_PTR va, VECTOR3D_PTR vscaled);
float VECTOR3D_Dot(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Cross(VECTOR3D_PTR va,VECTOR3D_PTR vb,VECTOR3D_PTR vn);
VECTOR3D VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb);
float VECTOR3D_Length(VECTOR3D_PTR va);
float VECTOR3D_Length_Fast(VECTOR3D_PTR va);
void VECTOR3D_Normalize(VECTOR3D_PTR va);
void VECTOR3D_Normalize(VECTOR3D_PTR va, VECTOR3D_PTR vn);
void VECTOR3D_Build(VECTOR3D_PTR init, VECTOR3D_PTR term,
                    VECTOR3D_PTR result);
float VECTOR3D_CosTh(VECTOR3D_PTR va, VECTOR3D_PTR vb);
void VECTOR3D_Print(VECTOR3D_PTR va, char *name);

// 4D 向量函数
void VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vsum);
VECTOR4D VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vdiff);
VECTOR4D VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Scale(float k, VECTOR4D_PTR va);
void VECTOR4D_Scale(float k, VECTOR4D_PTR va, VECTOR4D_PTR vscaled);
float VECTOR4D_Dot(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Cross(VECTOR4D_PTR va,VECTOR4D_PTR vb,VECTOR4D_PTR vn);
VECTOR4D VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb);
float VECTOR4D_Length(VECTOR4D_PTR va);
float VECTOR4D_Length_Fast(VECTOR4D_PTR va);
void VECTOR4D_Normalize(VECTOR4D_PTR va);
void VECTOR4D_Normalize(VECTOR4D_PTR va, VECTOR4D_PTR vn);
void VECTOR4D_Build(VECTOR4D_PTR init, VECTOR4D_PTR term,
         VECTOR4D_PTR result);
float VECTOR4D_CosTh(VECTOR4D_PTR va, VECTOR4D_PTR vb);
void VECTOR4D_Print(VECTOR4D_PTR va, char *name);

// 2×2 矩阵函数
void Mat_Init_2×2(MATRIX2X2_PTR ma,
        float m00, float m01, float m10, float m11);
void Print_Mat_2×2(MATRIX2X2_PTR ma, char *name);
float Mat_Det_2×2(MATRIX2X2_PTR m);
```

```
void Mat_Add_2x2(MATRIX2X2_PTR ma, MATRIX2x2_PTR mb, MATRIX2x2_PTR msum);
void Mat_Mul_2x2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2x2_PTR mprod);
int Mat_Inverse_2X2(MATRIX2x2_PTR m, MATRIX2X2_PTR mi);
int Solve_2X2_System(MATRIX2x2_PTR A, MATRIX1x2_PTR X, MATRIX1x2_PTR B);

// 3x3 矩阵函数

// 来自 T3DLIB1.CPP|H
int Mat_Mul_1x2_3x2(MATRIX1X2_PTR ma,
          MATRIX3X2_PTR mb,
          MATRIX1X2_PTR mprod);

// 来自 T3DLIB1.CPP|H
int Mat_Mul_1x3_3x3(MATRIX1X3_PTR ma,
          MATRIX3x3_PTR mb,
          MATRIX1x3_PTR mprod);

// 来自 T3DLIB1.CPP|H
int Mat_Mul_3x3(MATRIX3X3_PTR ma,
        MATRIX3x3_PTR mb,
        MATRIX3x3_PTR mprod);

// 来自 T3DLIB1.CPP|H
inline int Mat_Init_3x2(MATRIX3X2_PTR ma,
            float m00, float m01,
            float m10, float m11,
            float m20, float m21);

void Mat_Add_3x3(MATRIX3X3_PTR ma, MATRIX3x3_PTR mb, MATRIX3x3_PTR msum);
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va, MATRIX3X3_PTR mb,
             VECTOR3D_PTR vprod);
int Mat_Inverse_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mi);
void Mat_Init_3X3(MATRIX3X3_PTR ma,
            float m00, float m01, float m02,
            float m10, float m11, float m12,
            float m20, float m21, float m22);
void Print_Mat_3x3(MATRIX3x3_PTR ma, char *name);
float Mat_Det_3x3(MATRIX3x3_PTR m);
int Solve_3x3_System(MATRIX3x3_PTR A, MATRIX1x3_PTR X, MATRIX1x3_PTR B);

// 4x4 矩阵函数
void Mat_Add_4x4(MATRIX4x4_PTR ma, MATRIX4X4_PTR mb, MATRIX4x4_PTR msum);
void Mat_Mul_4x4(MATRIX4x4_PTR ma, MATRIX4X4_PTR mb, MATRIX4x4_PTR mprod);
void Mat_Mul_1x4_4X4(MATRIX1x4_PTR ma, MATRIX4x4_PTR mb,
          MATRIX1x4_PTR mprod);
void Mat_Mul_VECTOR3D_4X4(VECTOR3D_PTR va, MATRIX4x4_PTR mb,
             VECTOR3D_PTR vprod);
void Mat_Mul_VECTOR3D_4x3(VECTOR3D_PTR va, MATRIX4x3_PTR mb,
             VECTOR3D_PTR vprod);
void Mat_Mul_VECTOR4D_4x4(VECTOR4D_PTR va, MATRIX4x4_PTR mb,
             VECTOR4D_PTR vprod);
void Mat_Mul_VECTOR4D_4x3(VECTOR4D_PTR va, MATRIX4x4_PTR mb,
             VECTOR4D_PTR vprod);
int Mat_Inverse_4x4(MATRIX4X4_PTR m, MATRIX4x4_PTR mi);
void Mat_Init_4x4(MATRIX4X4_PTR ma,
            float m00, float m01, float m02, float m03,
            float m10, float m11, float m12, float m13,
            float m20, float m21, float m22, float m23,
            float m30, float m31, float m32, float m33);
void Print_Mat_4X4(MATRIX4X4_PTR ma, char *name);

// 四元数函数
void QUAT_Add(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qsum);
void QUAT_Sub(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qdiff);
void QUAT_Conjugate(QUAT_PTR q, QUAT_PTR qconj);
void QUAT_Scale(QUAT_PTR q, float scale, QUAT_PTR qs);
void QUAT_Scale(QUAT_PTR q, float scale);
```

```
float QUAT_Norm(QUAT_PTR q);
float QUAT_Norm2(QUAT_PTR q);
void QUAT_Normalize(QUAT_PTR q, QUAT_PTR qn);
void QUAT_Normalize(QUAT_PTR q);
void QUAT_Unit_Inverse(QUAT_PTR q, QUAT_PTR qi);
void QUAT_Unit_Inverse(QUAT_PTR q);
void QUAT_Inverse(QUAT_PTR q, QUAT_PTR qi);
void QUAT_Inverse(QUAT_PTR q);
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod);
void QUAT_Triple_Product(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR q3,
          QUAT_PTR qprod);
void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v, float theta);
void VECTOR4D_Theta_To_QUAT(QUAT_PTR q, VECTOR4D_PTR v, float theta);
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z, float theta_y,
         float theta_x);
void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v, float *theta);
void QUAT_Print(QUAT_PTR q, char *name);

// 2D 参数化直线函数
void Init_Parm_Line2D(POINT2D_PTR p_init, POINT2D_PTR p_term,
         PARMLINE2D_PTR p);
void Compute_Parm_Line2D(PARMLINE2D_PTR p, float t, POINT2D_PTR pt);
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
          float *t1, float *t2);
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
          POINT2D_PTR pt);

// 3D参数化直线函数
void Init_Parm_Line3D(POINT3D_PTR p_init, POINT3D_PTR p_term,
         PARMLINE3D_PTR p);
void Compute_Parm_Line3D(PARMLINE3D_PTR p, float t, POINT3D_PTR pt);

// 3D 平面函数
void PLANE3D_Init(PLANE3D_PTR plane, POINT3D_PTR p0,
        VECTOR3D_PTR normal, int normalize);
float Compute_Point_In_Plane3D(POINT3D_PTR pt, PLANE3D_PTR plane);
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline, PLANE3D_PTR plane,
                float *t, POINT3D_PTR pt);

// 定点数函数
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2);
FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2);
void FIXP16_Print(FIXP16 fp);
```

函数是不是很多？如前面所述，这个数学引擎支持编写 3D 游戏所需的全部功能。必要时我们将对其进行优化，但就现在而言，这是一套非常好的数学函数。我迫不及待地想介绍它们是如何工作的。

5.6 全局变量

数学引擎中的全局变量不多，实际上，仅有的几个全局变量都来自 T3DLIB1.CPP|H 中最基本的数学支持。这些全局变量是正弦和余弦查找表，如下所示：

```
// 用于存储查找表，来自 T3DLIB1.CPP|H
extern float cos_look[361]; // 以便能够存储 0-360 度的值
extern float sin_look[361]; // 以便能够存储 0-360 度的值
```

注意，这些表以度（而不是弧度）为单位，包含 361 个元素，分别代表[0...360]度。您可能会说，360 度与 0 度是一回事，确实如此，但添加这个元素可简化某些算法的编写工作，且节省了一次比较操作。要初始化这些表，只需在应用程序开头调用函数 Build_Sin_Cos_Tables()。

注意：在更健壮的数学引擎中，可能有很多全局状态变量（它们记录数学引擎的状态）和更复杂的数据结构，如矩阵栈（如 D3D 和 OpenGL 的 API 使用的矩阵栈）。

5.7　数学引擎 API 清单

下面介绍整个数学引擎 API（它位于 T3DLIB4.CPP|H 文件中）。对于每个函数，将介绍其功能并提供一个简单的调用范例；对于有些比较复杂的数学 API 函数，还将提供演示程序。另外，对于很多代表性的函数，将列出其代码，让读者知道是如何编写的。最后，对于位于 T3DLIB1.CPP|H 而不是 T3DLIB4.CPP|H 中的函数，读者可在第 3 章找到它们的描述。

5.7.1　三角函数

函数原型：

```
float Fast_Sin(float theta);

float Fast_Cos(float theta);
```

函数代码：

```
float Fast_Sin(float theta)
{
// 这个函数使用查找表 sin_look[]
// 但能够通过插值计算负角度和小数角度的正弦
// 因此与查找相比，精度更高，但速度可能低些

// 将角度转换为 0-359 的值
theta = fmodf(theta,360);
// 将角度转换为正值
if (theta < 0) theta+=360.0;

// 提取角度的整数部分和小数部分，以便进行插值计算
int theta_int  = (int)theta;
float theta_frac = theta - theta_int;

// 使用查找表并根据小数部分进行插值来计算正弦
// 如果 theta_int 为 359，则加 1 后将为 360
// 但这没有关系，因为查找表中包含 360 度的正弦值
return(sin_look[theta_int] +
    theta_frac*(sin_look[theta_int+1] - sin_look[theta_int]));

} // end Fast_Sin
```

用途：

这两个函数分别使用查找表 sin_look[ ]和 cos_look[ ]并通过线性插值来计算参数的正弦值和余弦值。与仅使用查找表相比，它们的精度更高；同时与使用 C/C++内部数学库函数 sin() 和 cos() 相比，速度更快。参数 theta 必须以度为单位。图 5.11 说明了该函数如何使用查找表进行插值。

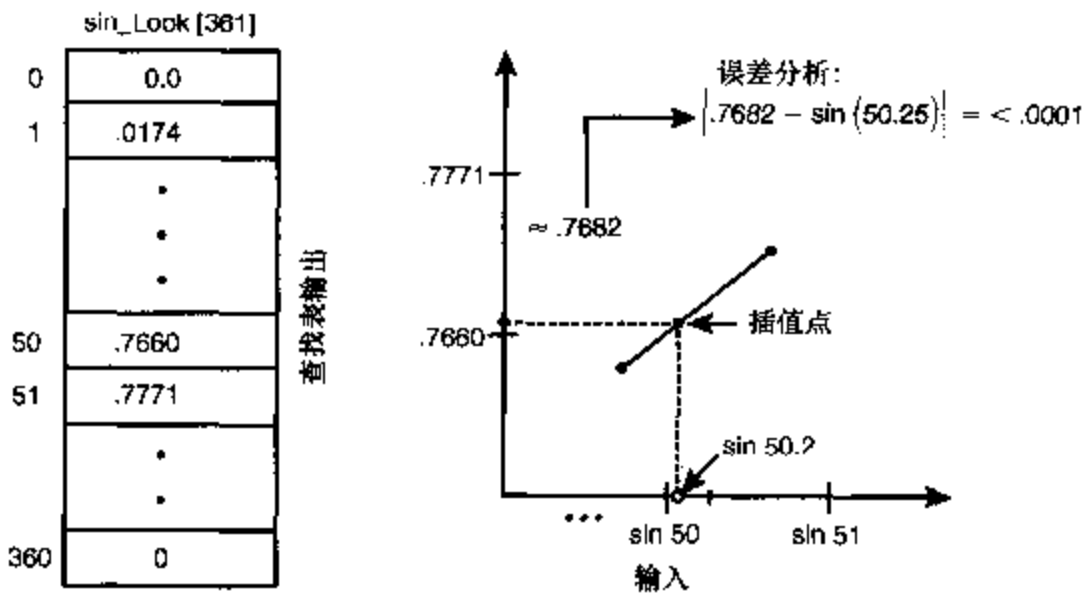


图 5.11　通过插值提高三角函数查找表的精度

范例：

下面的代码演示了如何计算 50.2 度的正弦和余弦。

```
// 首先创建查找表
Build_Sin_Cos_Tables();

float answer1 = Fast_Sin(50.2)
float answer2 = Fast_Cos(50.2)
```

5.7.2 坐标系支持函数

这部分 API 函数处理笛卡尔坐标与极坐标、柱面坐标和球面坐标之间的转换。每个函数都是按照第 4 章介绍的数学理论来实现的，因此没有必要列出其代码，除非您想了解如何将类似下面的公式：

```
x=r*sin(theta)
```

转换为 C/C++语句：

```
x=r*sin(theta);
```

另外，这些函数的返回类型都是 void。下面列出这些函数。

函数原型：

```
void POLAR2D_To_POINT2D(POLAR2D_PTR polar, POINT2D_PTR rect);
```

用途：

void POLAR2D_To_POINT2D()将一个用 r 和 theta 表示的 2D 极坐标点转换为一个 (x, y) 点，并将它存储在一个 POINT2D 结构中。

范例：

```
POLAR2D pp = {5, PI2};
POINT2D pr = {0,0};

// 将极坐标转换为迪卡尔坐标
POLAR2D_To_POINT2D(&pp, &pr);
```

函数原型：

```
void POLAR2D_To_RectXY(POLAR2D_PTR polar, float *x, float *y);
```

用途：

void POLAR2D_To_RectXY()将一个 2D 极坐标点转换为 x、y 坐标。

范例：

```
POLAR2D pp = {5, PI2};
float x=0, y=0;

// 将极坐标转换为 x、y 坐标
POLAR2D_To_RectXY(&pp, &x, &y)
```

函数原型：

```
void POINT2D_To_POLAR2D(POINT2D_PTR rect, POLAR2D_PTR polar);
```

用途：

void POINT2D_To_POLAR2D()将一个用直角坐标表示的点转换为 2D 极坐标格式。

范例：

```
POLAR2D pp = {0, 0};
POINT2D pr = {10,20};

// 将直角坐标转换为极坐标
POINT2D_To_POLAR2D(&pr, &pp);
```

函数原型：

```
void POINT2D_To_PolarRTh(POINT2D_PTR rect, float *r, float *theta);
```

用途：

void POINT2D_To_PolarRTh()将一个 2D 极坐标点转换为 *r* 和 theta。

范例：

```
POLAR2D pp = {3, PI};
float r =0, theta = 0;

// 转换为 r 和 theta
POINT2D_To_PolarRTh(&pp, &r, &theta);
```

函数原型：

```
void CYLINDRICAL3D_To_POINT3D(CYLINDRICAL3D_PTR cyl, POINT3D_PTR rect);
```

用途：

void CYLINDRICAL3D_To_POINT3D()将一个 3D 柱面坐标点转换为一个 3D 直角坐标点。

范例：

```
CYLINDRICAL3D pc = {10, PI/5, 20}; // r、theta、z
POINT3D pr = {0,0,0};

// 将柱面坐标转换为直角坐标
CYLINDRICAL3D_To_POINT3D(&pc, &pr);
```

函数原型：

```
void CYLINDRICAL3D_To_RectXYZ(CYLINDRICAL3D_PTR cyl, float *x, float *y, float *z);
```

用途：

void CYLINDRICAL3D_To_RectXYZ()将一个 3D 柱面坐标点转换为 *x*、*y*、*z* 坐标。

范例：

```
CYLINDRICAL3D pc = {10, PI/5, 20}; // r、theta、z
float x=0, y=0, z=0;

// 将柱面坐标点转换为 x、y、z 坐标值
CYLINDRICAL3D_To_RectXYZ(&pc, &x, &y, &z);
```

函数原型：

```
void POINT3D_To_CYLINDRICAL3D(POINT3D_PTR rect, CYLINDRICAL3D_PTR cyl);
```

用途：

void POINT3D_To_CYLINDRICAL3D()将 3D 直角坐标点转换为 3D 柱面坐标点。

范例：

```
CYLINDRICAL3D pc = {0,0,0}; // r、theta、z
POINT3D pr = {1,2,3};

// 将直角坐标点转换为柱面坐标点
POINT3D_To_CYLINDRICAL3D(&pr, &pc);
```

函数原型：

```
void POINT3D_To_CylindricalRThZ(POINT3D_PTR rect, float *r,
                float *theta, float *z);
```

用途：

void POINT3D_To_CylindricalRThZ()将 3D 点转换为柱面坐标 *r*、theta、*z*。

范例：

```
POINT3D pr = {1,2,3};
float r=0, theta=0, z=0;

// 将 3D 点转换为柱面坐标值
POINT3D_To_CylindricalRThZ(&pr, &r, &theta, &z);
```

函数原型：

```
void SPHERICAL3D_To_POINT3D(SPHERICAL3D_PTR sph, POINT3D_PTR rect);
```

用途：

void SPHERICAL3D_To_POINT3D()将一个 3D 球面坐标点转换为一个 3D 直角坐标点。

范例：

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p、theta、phi
POINT3D pr = {0,0,0};

// 将球面坐标点转换为直角坐标点
SPHERICAL3D_To_POINT3D(&ps, &pr);
```

函数原型：

```
void SPHERICAL3D_To_RectXYZ(SPHERICAL3D_PTR sph,
              float *x, float *y, float *z);
```

用途：

void SPHERICAL3D_ToRectXYZ()将一个 3D 球面坐标点转换为 x、y、z 坐标。

范例：

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p、theta、phi
float x=0, y=0, z=0;

// 将球面坐标点转换为直角坐标值
SPHERICAL3D_To_RectXYZ(&ps, &x, &y, &z);
```

函数原型：

```
void POINT3D_To_SPHERICAL3D(POINT3D_PTR rect, SPHERICAL3D_PTR sph);
```

用途：

void POIND3D_To_SPHERICAL3D()将一个 3D 点转换为球面坐标点。

范例：

```
SPHERICAL3D ps = {1, PI/4, PI/2}; // p、theta、phi
POINT3D pr = {0,0,0};

// 将 3D 点转换为球面坐标点
POINT3D_To_SPHERICAL3D(&pr, &ps);
```

函数原型：

```
void POINT3D_To_SphericalPThPh(POINT3D_PTR rect, float *p,
                float *theta, float *phi);
```

用途：

void POINT_To_SphericalPThPh()将一个 3D 点转换为球面坐标 rho、theta 和 phi。

范例：

```
POINT3D pr = {10,20,30};
float p=0, theta=0, phi=0;

// 将 3D 点转换为球面坐标值
POINT3D_To_SphericalPThPh(&pr, &p, &theta, &phi);
```

5.7.3 向量支持函数

数学引擎提供了对三种向量的支持：2D、3D 和 4D。2D 和 3D 向量支持很容易理解，但 4D 向量需要稍微说明一下。还记得齐次坐标吗？如 **p** = <x, y, z, w>，要将 **p** 转换为实际的 3D 坐标 **p'**，可执行如下转换：

```
p' = p/w = <x/w, y/w, z/w, w/w>
```

然后丢弃最后一个分量：

```
p' = <x/w, y/w, z/w>
```

数学知识：很多 4D 向量运算都不考虑第 4 个分量 w，只对前面的<x，y，z>分量执行运算。

对 4D 向量就复习到这里。对于 2D、3D 和 4D 向量的各种运算，都提供了相应的向量函数。当然，有些操作并非在每种维数下都有意义，例如，2D 向量的叉积就没有定义。然而，在很大程度上说，所有向量函数都有 2D、3D 和 4D 版本，这里只列举一个版本的实例。另外，对于只在特定维数下才有意义的向量函数，将单独介绍，同时将类似的向量函数作为一组。另外，很多函数的返回值为浮点数或相应的向量类型；例如，将 2D、3D 和 4D 向量相加的函数分别返回一个 2D、3D 和 4D 向量。还有，将使用“*D”来表示 2D、3D 和 4D。

注意：点和向量的数据类型相同。因此，在任何使用 VECTOR3D 的位置，也可以使用 POINT3D。当然，对点执行某些向量运算可能没有意义，如计算两点的叉积。然而，创建向量时，可以使用两个点、两个向量、一个点或一个向量。

函数原型：

```
void VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vsum);

void VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vsum);

void VECTOR4D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vsum);
```

用途：

void VECTOR*D_Add()将指定向量 va 和 vb 相加，如图 5.12 所示；然后将结果存储在 vsum 中。

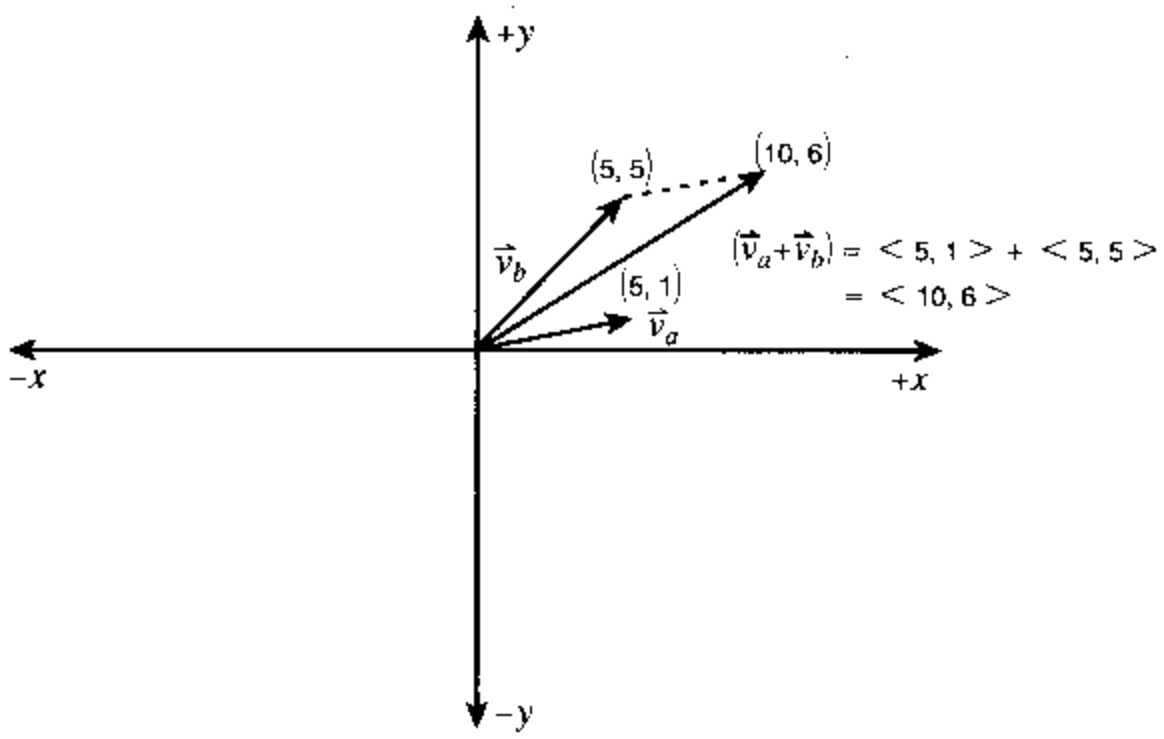


图 5.12　向量相加

范例：

```
VECTOR3D v1 = {1,2,3};
VECTOR3D v2 = {5,6,7};
VECTOR3D vsum; // 用于存储结果

// 将两个向量相加
VECTOR3D_Add(&v1, &v2, &vsum);
```

函数原型：

```
VECTOR2D VECTOR2D_Add(VECTOR2D_PTR va, VECTOR2D_PTR vb);

VECTOR3D VECTOR3D_Add(VECTOR3D_PTR va, VECTOR3D_PTR vb);

VECTOR4D VECTOR3D_Add(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

用途：

VECTOR*D VECTOR*D_Add()向量相加函数的堆栈版本，将结果返回到堆栈中。

范例：

```
VECTOR2D v1 = {1,2};
VECTOR2D v2 = {5,6};

// 相加并存储结果
VECTOR2D vsum = VECTOR2D_Add(&v1, &v2);
```

函数原型：

```
void VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb, VECTOR2D_PTR vdiff);

void VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb, VECTOR3D_PTR vdiff);

void VECTOR4D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb, VECTOR4D_PTR vdiff);
```

用途：

void VECTOR*D_Sub()将两个向量相减：(vb –va)，如图 5.13 所示；并将结果存储到 vdiff 中。

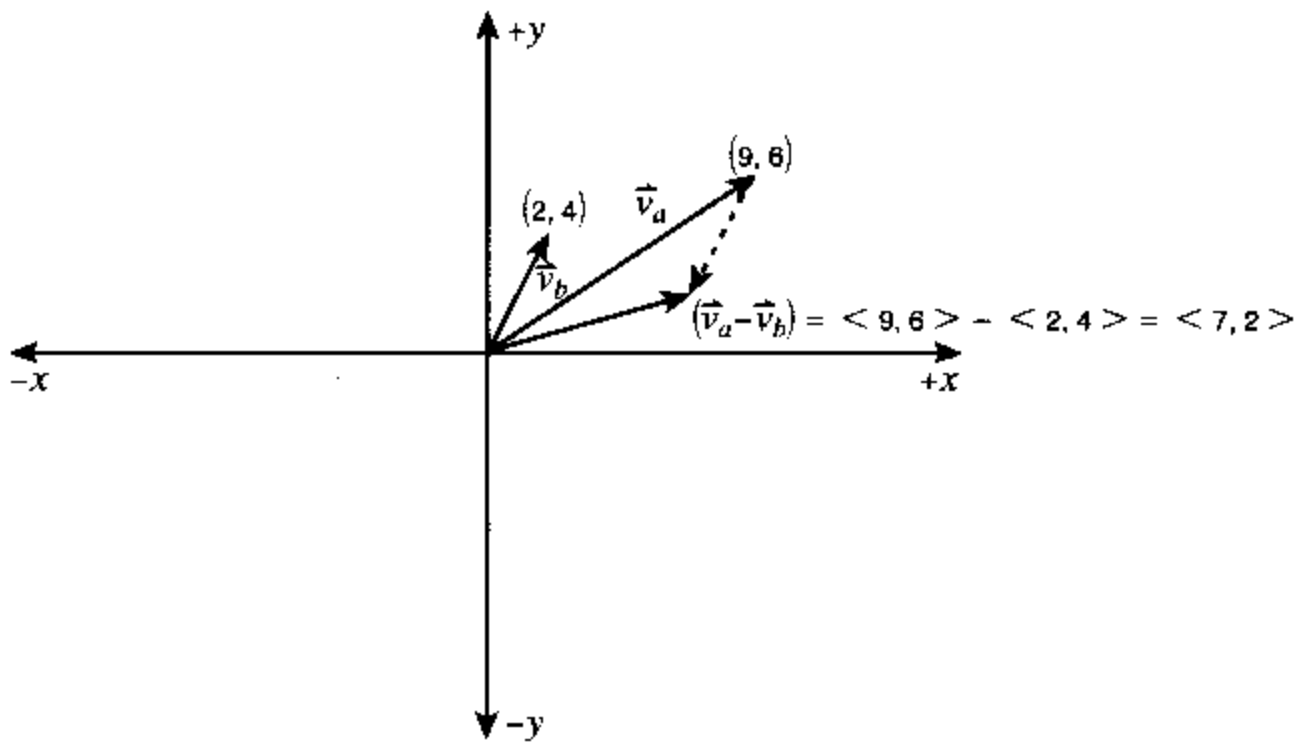


图 5.13　向量相减

范例：

```
VECTOR3D v1 = {1,2,3};
VECTOR3D v2 = {5,6,7};
VECTOR3D vdiff; // 用于存储结果

// 将向量相减
VECTOR3D_Sub(&v1, &v2, &vdiff);
```

函数原型：

```
VECTOR2D VECTOR2D_Sub(VECTOR2D_PTR va, VECTOR2D_PTR vb);

VECTOR3D VECTOR3D_Sub(VECTOR3D_PTR va, VECTOR3D_PTR vb);

VECTOR4D VECTOR3D_Sub(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

用途：

VECTOR*D VECTOR*D_Sub()是向量相减函数的堆栈版本，将结果返回到堆栈。

范例：

```
VECTOR2D v1 = {1,2};
VECTOR2D v2 = {5,6};

// 相减并存储结果
VECTOR2D vdiff = VECTOR2D_Sub(&v1, &v2);
```

函数原型：

```
void VECTOR2D_Scale(float k, VECTOR2D_PTR va, VECTOR2D_PTR vscaled);

void VECTOR3D_Scale(float k, VECTOR3D_PTR va, VECTOR3D_PTR vscaled);

void VECTOR4D_Scale(float k, VECTOR4D_PTR va, VECTOR4D_PTR vscaled);
```

用途：

void VECTOR*D_Scale() 使用缩放因子 k 对向量 va 进行缩放，如图 5.14 所示；然后将结果（k*va）存储在 vscaled 中。

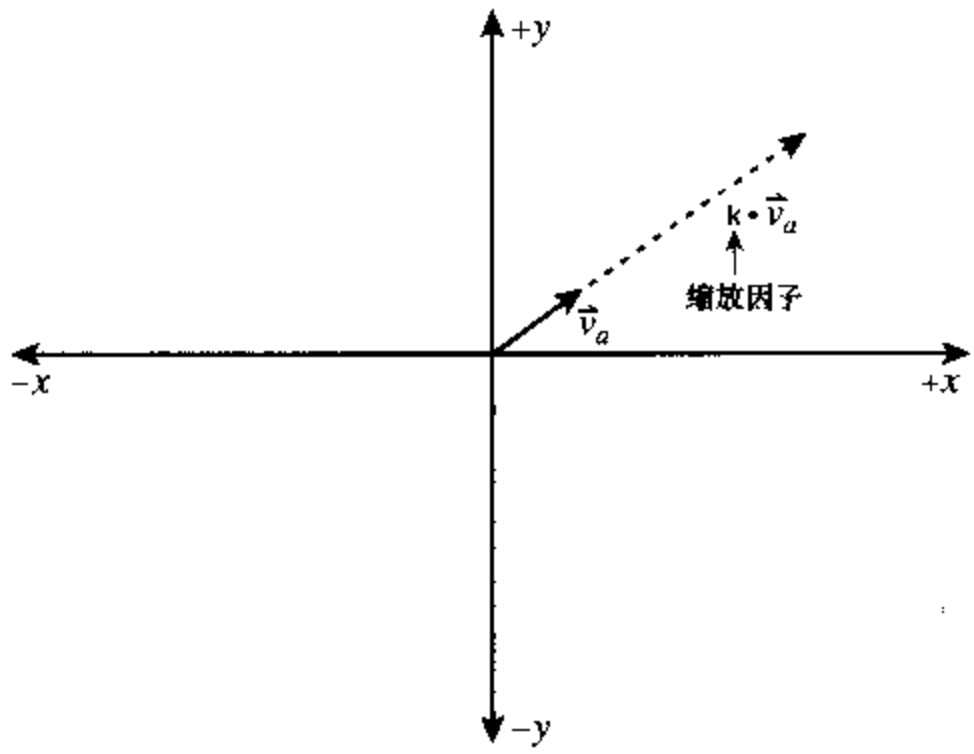


图 5.14　向量缩放

范例：

```
VECTOR3D v1 = {1,1,1};
VECTOR3D vs;

// 将 v1 放大 50 倍
VECTOR3D_Scale(50, &v1, &vs);

// 反转向量 v1 的方向
VECTOR3D_Scale(-1, &v1, &vs);
```

函数原型：

```
float VECTOR2D_Dot(VECTOR2D_PTR va, VECTOR2D_PTR vb);

float VECTOR3D_Dot(VECTOR3D_PTR va, VECTOR3D_PTR vb);

float VECTOR4D_Dot(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

用途：

float VECTOR*D_Dot() 计算点积 va.vb，并返回标量结果。注意，VECTOR4D 版本忽略 *w* 分量。

范例：

```
VECTOR4D v1={1,0,0,1}; // 与 x 轴平行的单位向量
VECTOR4D v2={0,1,0,1}; // 与 y 轴平行的单位向量

// 计算它们的点积，结果应为 0
float dp=VECTOR4D_Dot(&v1, &v2);
```

函数原型：

```
void VECTOR3D_Cross(VECTOR3D_PTR va,VECTOR3D_PTR vb,VECTOR3D_PTR vn);

void VECTOR4D_Cross(VECTOR4D_PTR va,VECTOR4D_PTR vb,VECTOR4D_PTR vn);
```

用途：

void VECTOR*D_Cross()计算向量叉积 va×vb，如图 5.15 所示；即计算与向量 va 和 vb 都垂直的向量，并将结果存储在 vn 中。VECTOR4D 版本忽略 w 分量。

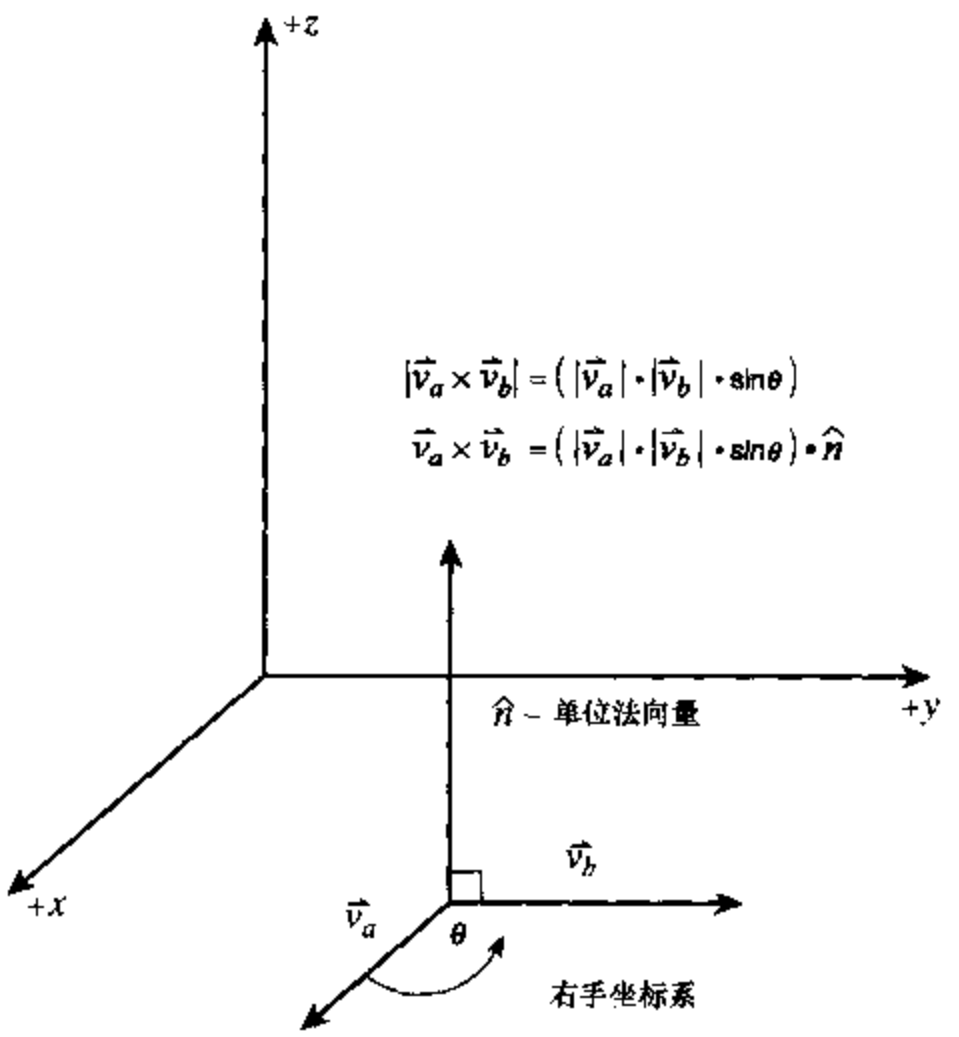


图 5.15　向量叉积

范例：

```
VECTOR3D vx = {1,0,0};
VECTOR3D vy = {0,1,0};
VECTOR3D vcross;

// 计算分别与 x 轴和 y 轴平行的单位向量的叉积，结果应为与 z 轴平行的单位向量
VECTOR3D_Cross(&vx, &vy, &vcross);
```

函数原型：

```
VECTOR3D VECTOR3D_Cross(VECTOR3D_PTR va, VECTOR3D_PTR vb);

VECTOR4D VECTOR4D_Cross(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

用途：

VECTOR*D VECTOR*D_Cross()计算向量叉积 va×vb，即与向量 va 和 vb 都垂直的向量，并将结果返回到堆栈中。VECTOR4D 版本忽略 w 分量。

范例：

```
VECTOR3D vx = {1,0,0};
VECTOR3D vy = {0,1,0};

// 计算分别与 x 轴和 y 轴平行的单位向量的叉积，结果应为与 z 轴平行的单位向量
VECTOR3D vcross = VECTOR3D_Cross(&vx, &vy);
```

函数原型：

```
float VECTOR2D_Length(VECTOR2D_PTR va);
float VECTOR3D_Length(VECTOR3D_PTR va);
float VECTOR4D_Length(VECTOR4D_PTR va);
```

用途：

float VECTOR*D_Length()使用标准的“平方和的平方根”算法计算指定向量的长度。

范例：

```
VECTOR2D v1={3,4};

// 计算长度，结果应为 5
float length = VECTOR3D_Length(&v1);
```

函数原型：

```
float VECTOR2D_Length_Fast(VECTOR2D_PTR va);

float VECTOR3D_Length_Fast(VECTOR3D_PTR va);

float VECTOR4D_Length_Fast(VECTOR4D_PTR va);
```

用途：

float VECTOR*D_Length_Fast()使用泰勒级数近似计算指定向量的长度。它的误差最多只有几个百分点，但速度比 VECTOR*D_Length()函数快 10 倍以上。

范例：

```
VECTOR2D v1={3,4};

// 计算长度，结果应为 5
float length = VECTOR3D_Length_Fast(&v1);
```

函数原型：

```
void VECTOR2D_Normalize(VECTOR2D_PTR va);

void VECTOR3D_Normalize(VECTOR3D_PTR va);

void VECTOR4D_Normalize(VECTOR4D_PTR va);
```

用途：

void VECTOR*D_Normalize()将向量 va 归一化；即将向量 va 的各个分量除以其长度，使之成为一个单位向量。注意，va 的值将被修改。

范例：

```
VECTOR2D v={5,6};

// 将 v 归一化：v = v/|v|
VECTOR2D_Normalize(&v);
```

函数原型：

```
void VECTOR2D_Normalize(VECTOR2D_PTR va, VECTOR2D_PTR vn);

void VECTOR3D_Normalize(VECTOR3D_PTR va, VECTOR3D_PTR vn);

void VECTOR4D_Normalize(VECTOR4D_PTR va, VECTOR4D_PTR vn);
```

用途：

void VECTOR2D_Normalize()将向量 va 归一化，并将结果存储在 vn 中。

范例：

```
VECTOR3D v={1,2,3};
VECTOR3D vn;

// vn = v/|vn|
VECTOR3D_Normalize(&v, &vn);
```

函数原型：

```
void VECTOR2D_Build(VECTOR2D_PTR init, VECTOR2D_PTR term, VECTOR2D_PTR result);

void VECTOR3D_Build(VECTOR3D_PTR init, VECTOR3D_PTR term, VECTOR3D_PTR result);

void VECTOR4D_Build(VECTOR4D_PTR init, VECTOR4D_PTR term, VECTOR4D_PTR result);
```

用途：

void VECTOR*D_Build()创建一个从 init 到 term 的向量，并将它存储到 result 中。这些函数的参数类型也可以是 POINT*D，因为大多数情况下您都想根据两个点来创建一个向量。

范例：

```
POINT3D p1={1,2,3}, p2={4,5,6};
VECTOR3D v12; // use to store the result

// 创建一个从 p1 到 p2 的向量
VECTOR3D_Build(&p1, &p2, &v);
```

注意，不需要执行强制类型转换，因为维数相同的点和向量的数据类型相同。然而，也可以使用两个向量来创建一个向量。

函数原型：

```
float VECTOR2D_CosTh(VECTOR2D_PTR va, VECTOR2D_PTR vb);

float VECTOR3D_CosTh(VECTOR3D_PTR va, VECTOR3D_PTR vb);

float VECTOR4D_CosTh(VECTOR4D_PTR va, VECTOR4D_PTR vb);
```

用途：

float VECTOR*D_CosTh()计算两个向量 va 和 vb 之间的夹角的余弦，如图 5.16 所示。

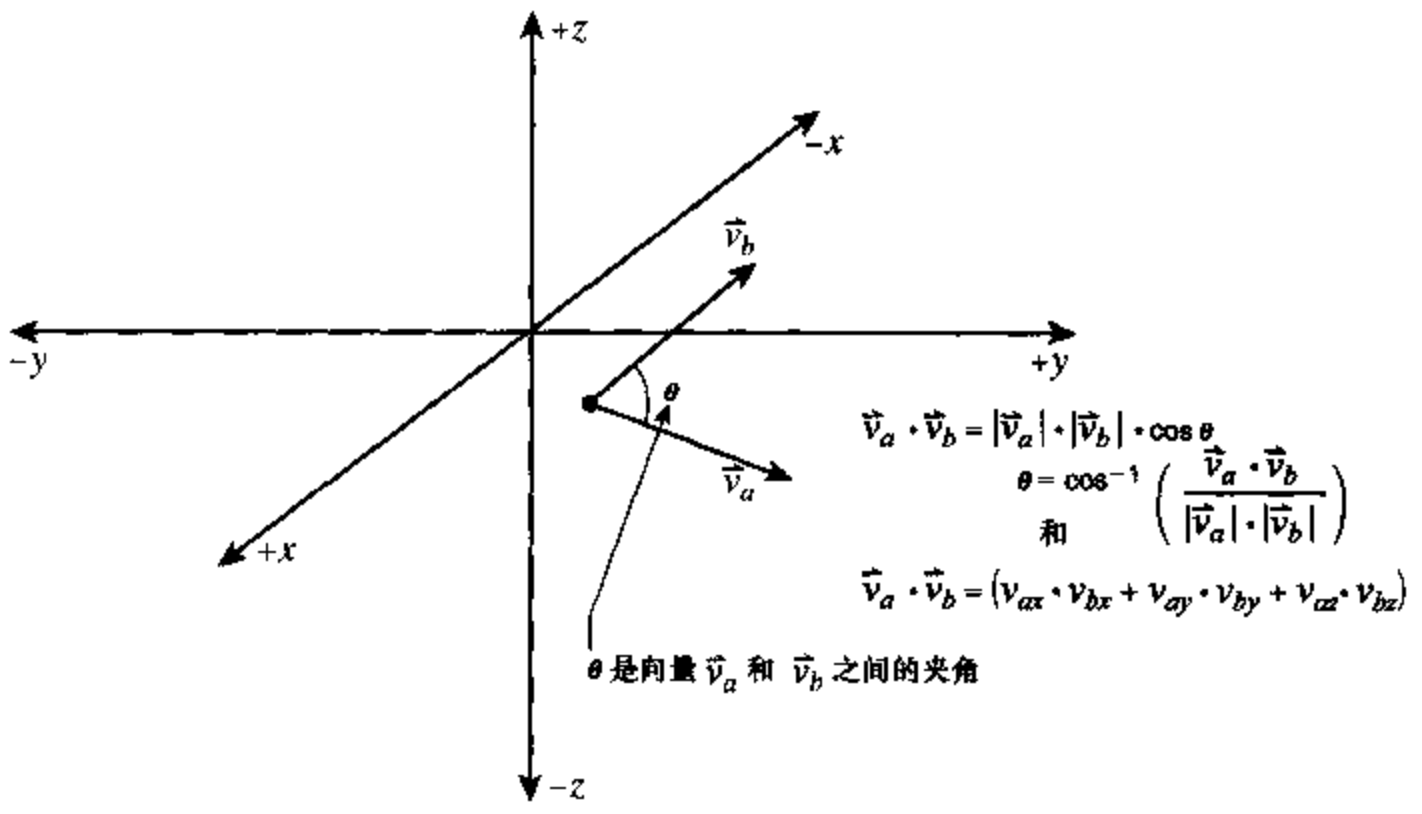


图 5.16　向量点积和夹角

在很多 3D 算法中，使用下述点积公式来计算角度θ（向量 u 和 v 之间的夹角）：

```
u.v = |u|*|v| * cos(θ)
```

将上述公式变换后，结果如下：

```
θ = cos⁻¹[(u . v)/(|u|*|v|)]
```

然而，很多时候并不需要知道真正的角度，只要其余弦就足够了：

```
cos(θ) = [(u . v)/(|u|*|v|)]
```

这就是关于该函数的全部说明。

提示：在游戏编程中，很多时候都可以利用数学理论来提高速度，如前例所示。另一个这样的例子是：假设有很多物体，当这些物体离黑洞的距离小于 100 个单位时，将被吸入黑洞中。在这种情况下，您可能盲目地使用距离函数和 sqrt( )函数来计算距离，如果稍微考虑一下，就会意识到没有必要使用实际的距离——

为什么不使用距离的平方呢？即使用 $d^2=(x^2+y^2+z^2)$，同时使用（100*100）而不是使用 100 作为判断变量。这样，就不需要使用 sqrt( )函数。其数学原理是：如果$|x|<|y|$，则$|x^2|<|y^2|$。但是注意不要溢出。

范例：

```
VECTOR3D u = {0,1,0}; // 与 y 轴平行的向量
VECTOR3D v = {1,1,0}; // 斜率为 45 的向量

// 计算 u 和 v 之间的夹角
VECTOR3D_CosTh(&u, &v);
```

注意：该函数对于隐藏面消除很有用。执行隐藏面消除时，需要判断观察方向与面法线之间的夹角为锐角还是钝角。

函数原型：

```
void VECTOR2D_Print(VECTOR2D_PTR va, char *name);

void VECTOR3D_Print(VECTOR3D_PTR va, char *name);

void VECTOR4D_Print(VECTOR4D_PTR va, char *name);
```

用途：

void VECTOR*D_Print()使用指定字符名以一种整齐的方式打印出向量。它通常作为一种调试功能，输出将写入到用 Open_Error_File()函数调用打开的错误文件中。

范例：

```
// 打开错误文件
Open_Error_File("error.txt");

VECTOR3D v={1,2,3};

// 打印向量
VECTOR3D_Print(&v, "Velocity");

// 关闭错误文件
Close_Error_File();
```

ERROR.TXT 文件的内容为：

```
Opening Error Output File (error.txt) on Sat Jan 29 22:36:00.520, 2000

Velocity=[1.000000, 2.000000, 3.000000, ]

Closing Error Output File.
```

5.7.4　矩阵支持函数

这套函数用于处理矩阵数学运算和矩阵变换。一般而言，3D 引擎将根据不同的需要使用 3×3 或 4×4 矩阵，但有时候也使用 4×3 矩阵，并对最后一列做出假设。我们有 3D 点和 4D 点数据结构，虽然有时候需要对矩阵数学进行很多优化，但在早期尽量一般化没什么坏处。另外，虽然我尽可能地考虑了可能需要的各种矩阵乘法和矩阵变换，但肯定会有遗漏——但以后可以添加它们。最后，虽然数学引擎支持 2×2、3×3 和 4×4 矩阵，但一些较老的 2×2 矩阵支持位于 T3DLIB1.CPP|H 模块中。

注意：矩阵库与向量库非常类似，它们都支持不同维数的数据类型，都使用类似的函数名，但它们也有非常大的差别，例如有些用于高维空间的函数的工作方式不同，因此本章分别介绍它们。这里将依次介绍用于 2×2、3×3 和 4×4 矩阵的函数。

函数原型：

```
void Mat_Init_2X2(MATRIX2X2_PTR ma, float m00, float m01, float m10, float m11);
```

用途：

void Mat_Init_2×2()使用指定的浮点数按先行后列的顺序初始化矩阵 ma。

范例：

```
MATRIX2X2 ma;

// 将矩阵初始化为单位矩阵
Mat_Init_2x2(&ma, 1, 0, 0, 1);
```

函数原型：

```
void Mat_Add_2x2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR msum);
```

用途：

void Mat_Add_2x2()将两个矩阵相加（ma + mb），并将结果存储到 msum 中。

范例：

```
MATRIX2X2 m1 = {1,2, 3,4};
MATRIX2X2 m2 = {4,9, 5,6};
MATRIX2X2 msum;

// 将矩阵相加
Mat_Add_2x2(&m1, &m2, &msum);
```

函数原型：

```
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR mprod);
```

函数代码：

```
void Mat_Mul_2X2(MATRIX2X2_PTR ma, MATRIX2X2_PTR mb, MATRIX2X2_PTR mprod)
{
// 这个函数将两个 2x2 矩阵相乘，并将结果存储到 mprod 中
mprod->M00 = ma->M00*mb->M00 + ma->M01*mb->M10;
mprod->M01 = ma->M00*mb->M01 + ma->M01*mb->M11;

mprod->M10 = ma->M10*mb->M00 + ma->M11*mb->M10;
mprod->M11 = ma->M10*mb->M01 + ma->M11*mb->M11;

} // end Mat_Mul_2x2
```

用途：

void Mat_Mul_2x2()将两个矩阵相乘（ma * mb），并将结果存储到 mprod 中。请参考前面的代码，这里将矩阵显式地相乘，而不使用循环操作。这种方法更优化。

范例：

```
MATRIX2X2 m1 = {1,2, 3,4};
MATRIX2X2 m2 = {1,0, 0,1};
MATRIX2X2 mprod;


// 将它们相乘，由于 m2 为单位矩阵，因此 m1*m2 = m1
Mat_Mul_2X2(&m1, &m2, &mprod);
```

函数原型：

```
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
          MATRIX3X2_PTR mb,
          MATRIX1X2_PTR mprod);
```

函数代码：

```
int Mat_Mul_1X2_3X2(MATRIX1X2_PTR ma,
          MATRIX3X2_PTR mb,
          MATRIX1X2_PTR mprod)
{
// 这个函数将一个 1x2 矩阵与一个 3x2 矩阵相乘，并存储结果
// 这里假设 1x2 矩阵的第三个元素为 1，使这种矩阵乘法合法，即变成 1x3 X 3x2

  for (int col=0; col<2; col++)
    {
    // 计算 ma 中的行与 mb 中的列之间的点积
```

```
    float sum = 0; // 用于存储结果

    for (int index=0; index<2; index++)
       {
       // 累积下一对元素的乘积
       sum+=(ma->M[index]*mb->M[index][col]);
       } // end for index
     // 累积最后一个元素与 1 的乘积
     sum+= mb->M[index][col];

     // 将结果存储到第 col 个元素中
     mprod->M[col] = sum;

     } // end for col

return(1);

} // end Mat_Mul_1X2_3X2
```

用途：

int Mat_Mul_1×2_3×2()是一个专用函数，将一个 1×2 矩阵（实质上是 2D 点）与一个 3×3 矩阵（表示旋转或平移）相乘。这种运算在数学上是未定义的，因为它们的内维不相同。然而，如果假设 1×2 实际是一个 1×3 矩阵，其最后一个元素为 1，将可以执行这种乘法。图 5.17 说明了执行该乘法的过程。另外，请阅读代码清单。对 2D 点进行变换时该函数很有用，您可能想节省空间，但同时支持平移和齐次坐标。

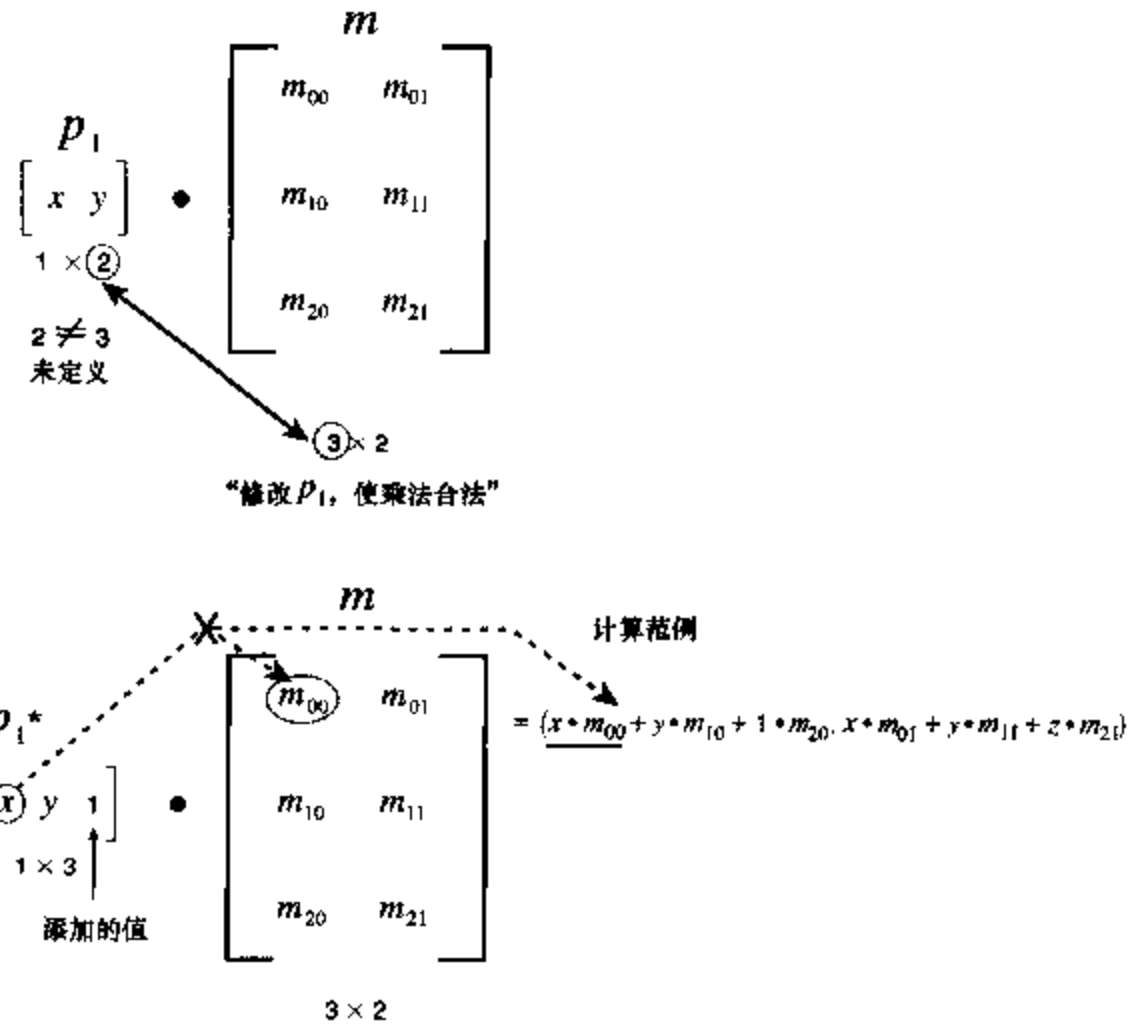


图 5.17　未定义的向量-矩阵相乘(1×2)*(3×2)

范例：

```
MATRI×1×2 p1={5,5}, p2; // 点

// 变换矩阵：旋转加平移
MATRI×3×2 m = {cos(th), sin(th),
       -sin(th), cos(th),
       dx,  dy};

// 执行矩阵乘法
Mat_Mul_1×2_3×2(&p1, &m, &p2);
```

提示：这个函数本身的速度非常慢。可以通过消除循环来提高加速，但是现在这样更清晰，读者可以看出其循环本质。必要时才进行优化，对代码优化之前应先对算法进行优化。有了最优化的算法和代码后，如果确实需要，可以使用内联汇编语言。

函数原型：

```
float Mat_Det_2×2(MATRIX2×2_PTR m);
```

用途：

float Mat_Det_2×2()用于计算矩阵 m 的行列式，并将结果返回到堆栈。

范例：

```
MATRI×2×2 m = {1,2,4,8}; // 这个矩阵是奇异的

// 计算行列式
float det = Mat_Det_2×2(&m);
```

函数原型：

```
int Mat_Inverse_2×2(MATRI×2×2_PTR m, MATRI×2×2_PTR mi);
```

用途：

int Mat_Inverse_2×2()用于计算矩阵 *m* 的逆矩阵（如果有的话），并将结果存储到 mi 中。如果逆矩阵存在，该函数返回 1；否则返回 0，且 mi 未定义。

范例：

```
MATRIX2X2 mA = {1,2, 4, -3};
MATRIX2X2 mI;

// 计算 mA 的逆矩阵
if (Mat_Inverse_2X2(&mA, &mI))
  {
  // 存在逆矩阵...
  } // end if
else
  {
  // 逆矩阵不存在
  } // end else
```

函数原型：

```
void Print_Mat_2×2(MATRIX2X2_PTR ma, char *name);
```

用途：

void Print_Mat_2×2()使用 name 指定的字符串作为标识符，以整齐的格式打印矩阵。当然，所有输出都将进入用 Open_Error_File()函数打开的错误文件中。

范例：

```
MATRIX2X2 m = {1,2,3,4};

// 打开错误文件，输出到屏幕
Open_Error_File(" ", stdout);

Print_Mat_2X2(&m,"Matrix m");

// 关闭错误文件
Close_Error_File();
```

函数原型：

```
void Mat_Init_3X3(MATRIX3X3_PTR ma,
         float m00, float m01, float m02,
         float m10, float m11, float m12,
         float m20, float m21, float m22);
```

用途：

void Mat_Init_3×3()使用传入的浮点数值以先行后列的顺序初始化矩阵 ma。

范例：

```
MATRIX3X3 ma;

// 将矩阵初始化为单位矩阵
Mat_Init_3X3(&ma, 1,0,0, 0,1,0, 0,0,1);
```

函数原型：

```
inline int Mat_Init_3X2(MATRIX3X2_PTR ma,
          float m00, float m01,
          float m10, float m11,
          float m20, float m21);
```

用途：

void Mat_Init_3×2()使用传入的浮点数值以先行后列的顺序初始化矩阵 ma。

范例：

```
MATRIX3X2 ma;

// 将左上角的 2×2 子矩阵初始化为单位矩阵
// matrix to identity matrix
Mat_Init_3X2(&ma, 1,0, 0,1, 0,0);
```

函数原型：

```
void Mat_Add_3X3(MATRIX3X3_PTR ma, MATRIX3X3_PTR mb, MATRIX3X3_PTR msum);
```

用途：

void Mat_Add_3×3()将两个矩阵相加（ma + mb），并将结果存储到 msum 中。

范例：

```
MATRIX3X3 m1 = {1,2,3,4,5,6,7,8,9};
MATRIX3X3 m2 = {4,9,7, -1,5,6, 2,3,4};
MATRIX3X3 msum;

// 将它们相加
Mat_Add_3X3(&m1, &m2, &msum);
```

函数原型：

```
void Mat_Mul_VECTOR3D_3X3(VECTOR3D_PTR va, MATRIX3X3_PTR mb, VECTOR3D_PTR vprod);
```

用途：

void Mat_Mul_VECTOR3D_3×3()将 1×3 的行向量 va 与 3×3 矩阵 mb 相乘，并将结果存储到 1×3 的行向量 vprod 中，如图 5.18 所示。实质上，该函数执行点或向量与矩阵的乘法。

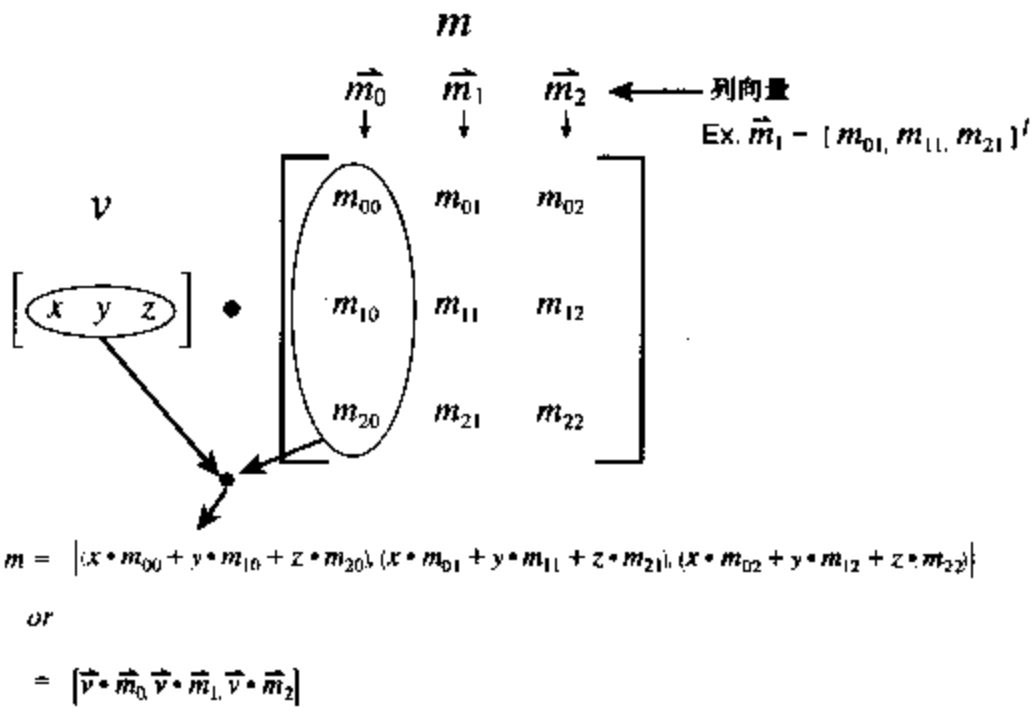


图 5.18　3D 向量与 3×3 矩阵相乘

范例：

```
VECTOR3D v={x,y,1}, vt;
MATRIX3X3 m = {1,0,0, 0,1,0,xt,yt,1};

// v*m
Mat_Mul_VECTOR3D_3X3(&v, &m, &vt);
```

读者知道这种变换的功能吗？如果假设 VECTOR3D 实际上是一个 2D 齐次点，则该变换执行如下平移：

```
vt.x = v.x+xt;
vt.y = v.y+yt;
vt.z = 1;
```

函数原型：

```
int (MATRIX1X3_PTR ma,
          MATRIX3X3_PTR mb,
          MATRIX1X3_PTR mprod);
```

用途：

int Mat_Mul_1×3_3×3()将一个 1×3 矩阵（实质上是一个行向量）与一个 3×3 矩阵相乘。该函数等价于函数 Mat_Mul_VECTOR3D_3×3()，只是类型不同。

范例：

```
MATRIX1X3 v={x,y,1}, vt;
MATRIX3X3 m = {1,0,0, 0,1,0,xt,yt,1};

// v*m
Mat_Mul_1X3_3X3(&v, &m, &vt);
```

函数原型：

```
int Mat_Mul_3X3(MATRIX3X3_PTR ma,
        MATRIX3X3_PTR mb,
        MATRIX3X3_PTR mprod);
```

用途：

void Mat_Mul_3×3()将两个矩阵相乘（ma * mb），并将结果存储到 mprod 中。

范例：

```
MATRIX3X3 m1 = {1,2,3, 4,5,6, 7,8,9};
MATRIX3X3 m2 = {1,0,0, 0,1,0, 0,0,1};
MATRIX3X3 mprod;

// 将它们相乘，由于 m2 为单位矩阵，因此 m1*m2 = m1
Mat_Mul_3X3(&m1, &m2, &mprod);
```

函数原型：

```
float Mat_Det_3X3(MATRIX3X3_PTR m);
```

用途：

float Mat_Det_3×3()计算矩阵 m 的行列式，并将结果返回到堆栈。

范例：

```
MATRIX3x3 m = {1,2,0, 4,8,9, 2,5,7};

// 计算行列式
float det = Mat_Det_3X3(&m);
```

函数原型：

```
int Mat_Inverse_3X3(MATRIX3X3_PTR m, MATRIX3X3_PTR mi);
```

用途：

int Mat_Inverse_3×3()用于计算矩阵 *m* 的逆矩阵（如果存在的话），并将结果存储到 mi 中。如果逆矩阵存在，该函数返回 1；否则返回 0，且 mi 未定义。

范例：

```
MATRIX3X3 mA = {1,2,9, 4,-3,6, 1,0,5};
MATRIX3X3 mI;

// 计算 mA 的逆矩阵
if (Mat_Inverse_3X3(&mA, &mI))
  {
  // 逆矩阵存在
  } // end if
else
  {
  // 逆矩阵不存在
  } // end else
```

函数原型：

```
void Print_Mat_3X3(MATRIX3X3_PTR ma, char *name);
```

用途：

void Print_Mat_3×3() 使用 name 指定的字符串作为标识符，以整齐的格式打印矩阵。当然，所有输出都将进入用 Open_Error_File() 函数打开的错误文件中。

范例：

```
MATRIX3X3 m = {1,2,3, 4,5,6, 7,8,9};

// 打开错误文件，并写入到该文件中
Open_Error_File("error.txt");
Print_Mat_3X3(&m, "Matrix m");

// 关闭错误文件
Close_Error_File();
```

函数原型：

```
void Mat_Init_4X4(MATRIX4X4_PTR ma,
      float m00, float m01, float m02, float m03,
      float m10, float m11, float m12, float m13,
      float m20, float m21, float m22, float m23,
      float m30, float m31, float m32, float m33);
```

用途：

void Mat_Init_4×4() 使用传入的浮点数值以先行后列的顺序初始化矩阵 ma。

范例：

```
MATRIX4X4 ma;

// 将矩阵初始化为单位矩阵
Mat_Init_4X4(&ma, 1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1);
```

函数原型：

```
void Mat_Add_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb, MATRIX4X4_PTR msum);
```

用途：

void Mat_Add_4×4() 将两个矩阵相加（ma + mb），并将结果存储到 msum 中。

范例：

```
MATRIX4X4 m1 = {1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16};
MATRIX4X4 m2 = {4,9,7,3, -1,5,6,7, 2,3,4,5, 2,0,5,3};
MATRIX4X4 msum;

// 将它们相加
Mat_Add_4X4(&m1, &m2, &msum);
```

函数原型：

```
void Mat_Mul_4X4(MATRIX4X4_PTR ma, MATRIX4X4_PTR mb, MATRIX4X4_PTR mprod);
```

用途：

void Mat_Mul_4×4()将两个矩阵相乘（ma * mb），并将结果存储到 mprod 中。

范例：

```
MATRIX4X4 m1 = {1,2,3,4, 4,5,6,7, 7,8,9,10, 11,12,13,14};
MATRIX4X4 m2 = {1,0,0,0, 0,1,0,0, 0,0,1,0, 0,0,0,1};
MATRIX4X4 mprod;

// 将它们相乘，由于 m2 为单位矩阵，因此 m1*m2 = m1
Mat_Mul_4X4(&m1, &m2, &mprod);
```

函数原型：

```
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma, MATRIX4X4_PTR mb, MATRIX1X4_PTR mprod);
```

函数代码：

```
void Mat_Mul_1X4_4X4(MATRIX1X4_PTR ma,
           MATRIX4X4_PTR mb,
           MATRIX1X4_PTR mprod)
{
// 这个函数将一个 1×4 矩阵与一个 4×4 矩阵相乘，并存储结果
// 这里没有做任何假设，而是直接相乘

  for (int col=0; col<4; col++)
    {
    // 计算 ma 中的行向量与 mb 中列向量的点积
    float sum = 0; // 用于存储结果

    for (int row=0; row<4; row++)
      {
      // 累积下一对元素的乘积
      sum+=(ma->M[row] * mb->M[row][col]);
      } // end for index

    // 将结果存储到第 col 个元素中
    mprod->M[col] = sum;

    } // end for col

} // end Mat_Mul_1X4_4X4
```

用途：

void Mat_Mul_1×4_4×4()将一个 1×4 的行向量与一个 4×4 矩阵相乘，并将结果存储到 mprod 中。该函数不做齐次坐标之类的假设，而是直接相乘，如代码清单所示。另外，通过显式地执行数学运算而不使用循环，可提高该函数的速度。

范例：

```
MATRIX1X4 v={x,y,z,1}, vt;
MATRIX4X4 m = {1,0,0,0 0,1,0,0, 0,0,1,0, xt,yt,zt,1};

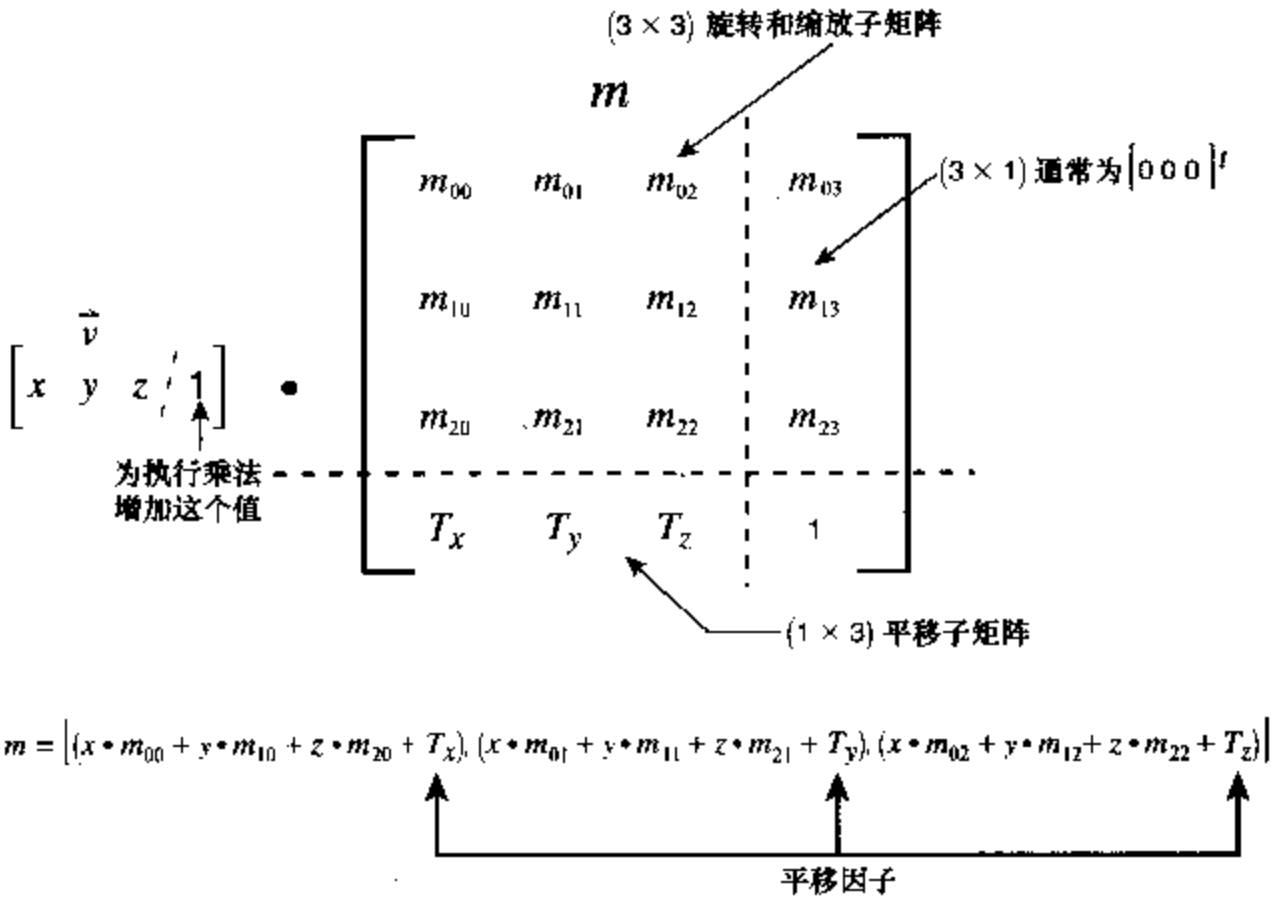
// v*m，相当于将 3D 点平移 x、y、z
Mat_Mul_1X4_4X4(&v, &m, &vt);
```

函数原型：

```
void Mat_Mul_VECTOR3D_4X4(VECTOR3D_PTR va, MATRIX4X4_PTR mb, VECTOR3D_PTR vprod);
```

用途：

void Mat_Mul_VECTOR3D_4×4 将一个 3D 向量与一个 4×4 矩阵相乘。然而，为执行这种乘法，该函数假设在向量 va 中存在虚拟的第 4 个元素，它等于 1.0。乘法执行后，结果是一个 3D 向量（而不是 4D 向量）。该函数用一个包含平移值的 4×4 矩阵对一个 3D 点进行变换，并假设 3D 点为齐次坐标。由于 3D 点只有三个元素[*x y z*]，没有空间存储齐次分量 *w*，所以假设它为 1.0。图 5.19 说明了这种乘法。



$$\vec{v} \cdot m = \left| (x \cdot m_{00} + y \cdot m_{10} + z \cdot m_{20} + T_x),\ (x \cdot m_{01} + y \cdot m_{11} + z \cdot m_{21} + T_y),\ (x \cdot m_{02} + y \cdot m_{12} + z \cdot m_{22} + T_z) \right|$$

图 5.19　3D 向量与 4×4 矩阵相乘

范例：

```
VECTOR3D v={10,10,10}, vrt;

// 绕 x 轴旋转并平移(tx,ty,tz)
MATRIX4X4 m = {1, 0, 0, 0,
        0, cos(th), sin(th),0,
        0,-sin(th), cos(th),0,
        tx, ty, tz, 1};

// 执行变换
Mat_Mul_VECTOR3D_4X4(&v, &m, &vrt);
```

可以将向量定义为点，并调用上述调用，因为 VECTOR3D 和 POINT3D 是相同的数据类型：

```
POINT3D p1={10,10,10}, p2;

// 绕 x 轴旋转并平移(tx,ty,tz)
MATRIX4X4 m = {1, 0, 0, 0,
        0, cos(th), sin(th),0,
        0,-sin(th), cos(th),0,
        tx, ty, tz, 1};

// 执行变换
Mat_Mul_VECTOR3D_4X4(&p1, &m, &p2);
```

函数原型：

```
void Mat_Mul_VECTOR3D_4X3(VECTOR3D_PTR va, MATRIX4X3_PTR mb,  VECTOR3D_PTR vprod);
```

用途：

void Mat_Mul_VECTOR3D_4×3()与函数 Mat_Mul_VECTOR3D_4×4 非常类似，但它是将一个 3D 向量与一个 4×3 矩阵（而不是 4×4 矩阵）相乘。它假设在向量 va 中存在第 4 个元素，且等于 1.0，以便执行乘法运算。由于在 4×3 矩阵中只有三列，所以结果自然为 3D 向量。

范例：

```
POINT3D p1={10,10,10}, p2;

// 绕 x 轴旋转并平移(tx,ty,tz)
// 这里不需要第 4 列，它总是[0 0 0 1]t
MATRIX4X3 m = {1, 0, 0,
```

```
        0, cos(th), sin(th),
        0,-sin(th), cos(th),
        tx, ty, tz};

// 执行变换
Mat_Mul_VECTOR3D_4X3(&p1, &m, &p2);
```

函数原型：

```
void Mat_Mul_VECTOR4D_4X4(VECTOR4D_PTR va, MATRIX4X4_PTR mb, VECTOR4D_PTR vprod);
```

用途：

void Mat_Mul_VECTOR_4×4()将 1×4 行向量 va 与 4×4 矩阵 mb 相乘，并将结果存储到 1×4 行向量 vprod 中。该函数不做任何假设，只是执行 1×4 矩阵和 4×4 矩阵乘法。

范例：

```
VECTOR4D v={10,10,10,1}, vrt;

// 绕 x 轴旋转并平移(tx,ty,tz)
MATRIX4X4 m = {1, 0, 0, 0,
        0, cos(th), sin(th),0,
        0,-sin(th), cos(th),0,
        tx, ty, tz, 1};

// do the transform
Mat_Mul_VECTOR4D_4X4(&v, &m, &vrt);
```

注意，结果为 vrt = <x', y', z', 1>。也就是说，齐次分量 w 等于 1.0。

函数原型：

```
void Mat_Mul_VECTOR4D_4X3(VECTOR4D_PTR va, MATRIX4X4_PTR mb, VECTOR4D_PTR vprod);
```

用途：

void Mat_Mul_VECTOR4D_4×3()与函数 Mat_Mul_VECTOR3D_4×3 非常类似，但它将一个 4D 向量（而不是 3D 向量）与一个 4×3 矩阵相乘。由于 4×3 矩阵只有三列，所以结果为一个 3D 向量，与 Mat_Mul_VECTOR3D_4×4 函数相同。最后，将 w 分量从向量 va 复制到 vprod 中；即假设 4×3 矩阵的最后一列为$[0\ 0\ 0\ 1]^t$。

范例：

```
POINT4D p1={10,10,10,1}, p2;

// 绕 x 轴旋转并平移(tx,ty,tz)
// 这里不需要第 4 列，它总是[0 0 0 1]^t
MATRIX4X3 m = {1, 0, 0,
        0, cos(th), sin(th),
        0,-sin(th), cos(th),
        tx, ty, tz};

// 执行变换
Mat_Mul_VECTOR4D_4X3(&p1, &m, &p2);
```

函数原型：

```
int Mat_Inverse_4X4(MATRIX4X4_PTR m, MATRIX4X4_PTR mi);
```

用途：

int Mat_Inverse_4×4()用于计算矩阵 m 的逆矩阵（如果存在的话），并将结果存储到 mi 中。如果逆矩阵存在，该函数返回 1；否则返回 0，且 mi 未定义。该函数只适用于最后一列为$[0\ 0\ 0\ 1]^t$的矩阵。

范例：

```
// 最后一列为[0 0 0 1]^t
MATRIX4X4 mA = {1, 2,9,0,
        4,-3,6,0,
        1, 0,5,0,
        2, 3,4,1};
```

```
MATRIX4X4 mI;

// 计算矩阵 mA 的逆矩阵
if (Mat_Inverse_4X4(&mA, &mI))
  {
  // 逆矩阵存在
  } // end if
else
  {
  // 逆矩阵不存在
  } // end else
```

函数原型：

```
void Print_Mat_4X4(MATRIX4X4_PTR ma, char *name);
```

用途：

void Print_Mat_4×4()使用 name 指定的字符串作为标识符，以整齐的格式打印矩阵。当然，所有输出都将进入用 Open_Error_File()函数打开的错误文件中。

范例：

```
MATRIX4X4 m = {1,2,3,4,
        5,6,7,8,
        9,10,11,12,
        13,14,15,16};

// 打开错误文件并写入到该文件
Open_Error_File("error.txt");

Print_Mat_4X4(&m,"Matrix m");

// 关闭错误文件
Close_Error_File();
```

5.7.5　2D 和 3D 参数化直线支持函数

我开始编写数学引擎时，根本没想到会在引擎中添加参数化直线支持。为什么呢？因为大多数时候，需要时可以在代码中使用一个向量和一个参数 t 手工编写参数化直线的代码。然而，既然大多数时候都将使用一个向量和一个参数 t，为什么不将它们封装到一个结构（或没有方法的类）中，然后以此为基础创建常用函数，如计算交点的函数呢？因此，数学引擎提供了对 2D 和 3D 参数化直线的支持以及一整套相关的函数。

为帮助回顾，下面再次给出 2D 参数化直线的结构：

```
typedef struct PARMLINE2D_TYP
{
POINT2D p0; // 参数化直线的起点
POINT2D p1; // 终点
VECTOR2D v; // 线段的方向向量
        // |v|=|p0->p1|
} PARMLINE2D, *PARMLINE2D_PTR;
```

这些值之间的关系如图 5.2 所示。注意，向量从 **p0** 到 **p1**，即没有归一化。下面是 3D 参数化直线的结构：

```
typedef struct PARMLINE3D_TYP
{
POINT3D p0; // 参数化直线的起点
POINT3D p1; // 终点
VECTOR3D v; // 线段的方向向量
        // |v|=|p0->p1|
} PARMLINE3D, *PARMLINE3D_PTR;
```

2D 版本和 3D 版本之间惟一的差别是 z 坐标；除此之外，它们以相同的格式存储相同的信息。下面分别介绍这些函数。

函数原型：

```
void Init_Parm_Line2D(POINT2D_PTR p_init, POINT2D_PTR p_term, PARMLINE2D_PTR p);
```

用途：

void Init_Parm_Line2D()根据指定的点计算它们之间的向量，并初始化一条 2D 参数化直线。注意，该向量是在函数内部生成的。

范例：

```
POINT2D p1 = {1,2}, p2 = {10,20};
PARMLINE2D p;

// 创建一条从 p1 到 p2 的参数化直线
Init_Parm_Line2D(&p1, &p2, &p);
```

函数原型：

```
void Compute_Parm_Line2D(PARMLINE2D_PTR p, float t, POINT2D_PTR pt);
```

用途：

void Compute_Parm_Line2D()计算 2D 参数化直线在参数 t 处的值，并将其返回到指定的点中。参数 $t=0$ 时，结果为起点 p1；参数 $t=1$ 时，结果为终点 p2。也就是说，当 t 从 0 变化到 1 时，将从 **p1** 移到 **p2**。

范例：

```
POINT2D p1 = {1,2}, p2 = {10,20}, pt;
PARMLINE2D p;

// 创建一条从 p1 到 p2 的参数化直线
Init_Parm_Line2D(&p1, &p2, &p);

// 计算 t=.5 对应的参数化直线上的点，并将其存储到 pt 中
Compute_Parm_Line2D(&p, 0.5, &pt);
```

函数原型：

```
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
                           float *t1, float *t2);
```

函数代码：

```
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2,
                           float *t1, float *t2)
{
// 这个函数计算两条参数化线段的交点
// 并将 t1 和 t2 分别设置交点在 p1 和 p2 上对应的 t 值
// 然而，t 值可能不在范围[0,1]内
// 这意味着线段本身没有相交，但它们对应的直线是相交的
// 函数的返回值为 0,表示没有相交，1 表示交点在线段上，
//2 表示相交，但交点不一定在线段上，3 表示两条线段位于同一条直线上

// 我们知道两条线段的参数化方程，需要计算交点（如果有的话）对应的 t1 和 t2

// 第 1 步：检测它们是否平行
// 如果一个方向向量是另一个向量与一个标量的乘积，则说明两条线段平行
// 除非它们重叠，否则不可能相交
float det_p1p2 = (p1->v.x*p2->v.y - p1->v.y*p2->v.x);
if (fabs(det_p1p2) <= EPSILON_E5)
  {
  // 这表明两条线段要么根本不相交，要么位于同一条直线上
  // 在后一种情况下，可能有一个或多个交点
  // 现在暂时假设它们不相交，以后需要时再重新编写该函数，以考虑重叠的情况
  return(PARM_LINE_NO_INTERSECT);

  } // end if

// 第 2 步：计算 t1 和 t2 的值
// 我们有两条以下述方式表示的线段
// p  = p0  + v*t
```

```
// p1  = p10  + v1*t1

// p1.x = p10.x + v1.x*t1
// p1.y = p10.y + v1.y*t1

// p2 = p20 + v2*t2

// p2.x = p20.x + v2.x*t2
// p2.y = p20.y + v2.y*t2
// 第 4 章介绍过如何计算交点
*t1 = (p2->v.x*(p1->p0.y - p2->p0.y) - p2->v.y*(p1->p0.x - p2->p0.x))
   /det_p1p2;

*t2 = (p1->v.x*(p1->p0.y - p2->p0.y) - p1->v.y*(p1->p0.x - p2->p0.x))
   /det_p1p2;

// 检查交点是否在线段上
if ((*t1>=0) && (*t1<=1) && (*t2>=0) && (*t2<=1))
  return(PARM_LINE_INTERSECT_IN_SEGMENT);
else
  return(PARM_LINE_INTERSECT_OUT_SEGMENT);

} // end Intersect_Parm_Lines2D
```

用途：

int Intersect_Parm_Lines2D()计算两条参数化直线 pl 和 p2 的交点，并将交点对应的参数值 t1 和 t2 分别存储到相应的变量中。根据两条直线是否相交以及交点是否在线段上，该函数有下列返回值。

```
#define PARM_LINE_NO_INTERSECT     0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
```

该函数不检测两条直线是否共线，因为这将极大地降低速度，同时有太多的情况（部分重叠、包含、只有一点重叠等）需要检测。因此，调用该函数前需要检查这些条件，或者在该函数中添加这些功能。请记住，这些参数化直线实际上是线段，同时非平行线总会在某一点相交，因此该函数将计算出交点。如果交点的 t 参数不在范围 0 <= *t* <= 1 内，则表明交点在线段 p1->p2 上。请参考附带光盘中的 DEMOII5_1.CPP|EXE；它允许用户定义参数化直线，并测试它们是否相交。

范例：

```
POINT2D p1 = {1,1}, p2 = {9,8};
POINT2D p3 = {2,8}, p4 = {7,1};

PARMLINE2D pl1, pl2;

// 创建一条从 p1 到 p2 的参数化直线
Init_Parm_Line2D(&p1, &p2, &pl1);

// 创建一条从 p3 到 p4 的参数化直线
Init_Parm_Line2D(&p3, &p4, &pl2);

float t1=0, t2=0; // 用于存储参数 t

// 计算交点
int intersection_type = Intersect_Parm_Lines2D(&pl1, &pl2, &t1, &t2);
```

函数原型：

```
int Intersect_Parm_Lines2D(PARMLINE2D_PTR p1, PARMLINE2D_PTR p2, POINT2D_PTR pt);
```

用途：

int Intersect_Parm_Lines2D()计算参数化直线的交点，但不返回交点对应的参数 t1 和 t2 的值，而是返回交点。当然，与前一个函数一样，您需要查看返回值来确认两条直线是否相交。

范例：

```
POINT2D p1 = {1,1}, p2 = {9,8};
POINT2D p3 = {2,8}, p4 = {7,1};

POINT2D pt; // 用于存储交点

PARMLINE2D pl1, pl2;

// 创建一条从 p1 到 p2 的参数化直线
Init_Parm_Line2D(&p1, &p2, &pl1);

// 创建一条从 p3 到 p4 的参数化直线
Init_Parm_Line2D(&p3, &p4, &pl2);

// 计算交点
int intersection_type = int Intersect_Parm_Lines2D(&pl1, &pl2, &pt);
```

C++: C 程序员可能会奇怪，为什么两个函数的名称相同。这是一项 C++功能。实质上，每个函数的参数类型是不同的；因此，在编译器看来而言，这两个函数是不同的。这种功能叫作函数重载，它根据参数列表来区分名称相同的函数。附录 D 将介绍 C++的这种功能。

下面介绍 3D 参数化直线函数。这里的函数更少，因为我将有些函数归类到了 3D 平面函数中。

函数原型：

```
void Init_Parm_Line3D(POINT3D_PTR p_init, POINT3D_PTR p_term, PARMLINE3D_PTR p);
```

用途：

void Init_Parm_Line3D()根据指定的点以及它们之间的向量，初始化一个 3D 参数化直线结构。注意，该向量是在函数内部生成的。

范例：

```
POINT3D p1 = {1,2,3}, p2 = {10,20,30};
PARMLINE3D p;

// 创建一条从 p1 到 p2 的参数化直线
Init_Parm_Line3D(&p1, &p2, &p);
```

函数原型：

```
void Compute_Parm_Line3D(PARMLINE3D_PTR p, float t, POINT3D_PTR pt);
```

用途：

void Compute_Parm_Line3D()计算一条参数化直线在参数 t 处的值，并将其返回存储到指定的点中。注意，参数 $t=0$ 时，结果为起点 p1；参数 $t=1$ 时，结果为终点 **p2**。也就是说，当 t 从 0 变化到 1 时，将从 **p1** 移到 **p2**。

范例：

```
POINT3D p1 = {1,2,3}, p2 = {10,20,30}, pt;
PARMLINE3D p;

// 创建一条从 p1 到 p2 的参数化直线
Init_Parm_Line3D(&p1, &p2, &p);

// 计算 t=.5 对应的参数化直线上的点
// 并将其存储到 pt 中
Compute_Parm_Line3D(&p, 0.5, &pt);
```

5.7.6 3D 平面支持函数

虽然有点太早，但我还是决定在数学引擎中添加对 3D 平面抽象实体的支持。实际上，在 99%以上的时间，我们处理的都是封闭多边形（位于同一个平面中），因此后面可能需要对这些函数进行改进。

现在，我们只要知道如何定义平面，然后使用平面执行操作即可。

我决定使用点-法线方式来表示平面，如图 5.20 所示。

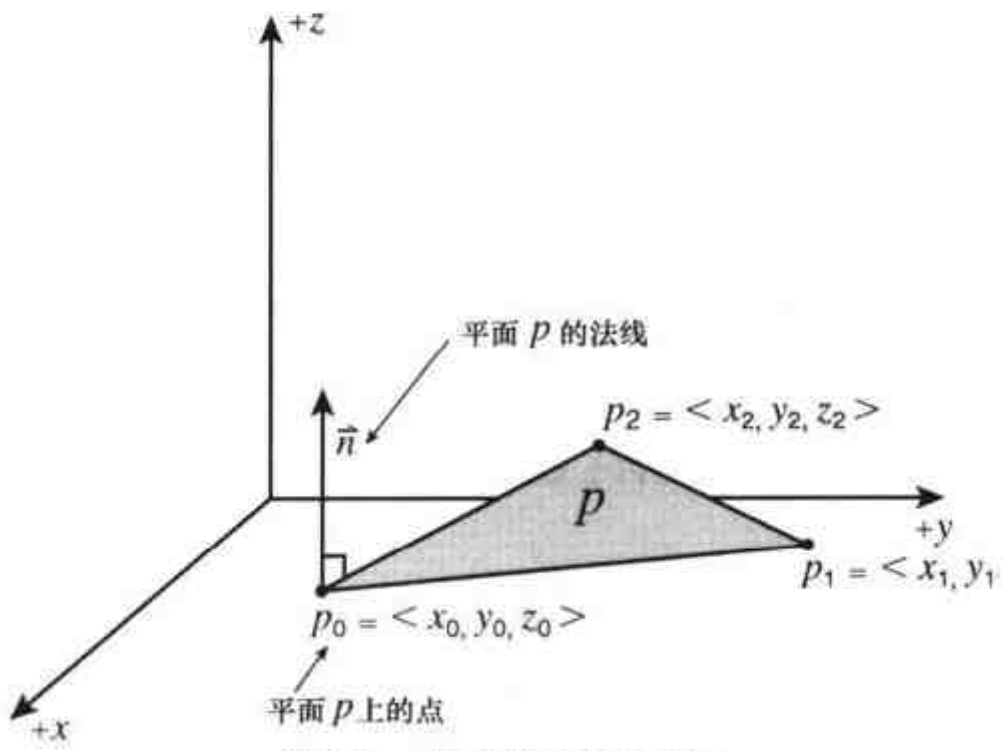


图 5.20　用点-法线定义 3D 平面

$nx*(x-x0) + ny*(y-y0) + nz*(z-z0) = 0$

其中 **n** = <nx，ny，nz>，**p0** = <x0，y0，z0>。

使用这种格式时，我们将平面存储为平面的法线和平面上的一个点。法线不必是单位向量。下面是 3D 平面的结构：

```
typedef struct PLANE3D_TYP
{
POINT3D p0; // 平面上的一个点
VECTOR3D n; // 平面的法线（不必是单位向量）
} PLANE3D, *PLANE3D_PTR;
```

同样，数学库中的很多函数旨在简化一些经常需要执行的重复性计算，或者简化数学对象或几何对象的表示。

函数原型：

```
void PLANE3D_Init(PLANE3D_PTR plane, POINT3D_PTR p0,
                  VECTOR3D_PTR normal, int normalize);
```

用途：

void PLANE3D_Init() 函数使用指定的点和法线来初始化一个 3D 平面。另外，该函数可以对指定的法线进行归一化，使其长度为 1.0。要进行归一化，需要将 normalize 参数设置为 TRUE；否则，将它设置为 FALSE。在很多光照和背面消除计算中，知道多边形或平面的法线长度为 1.0 会很有帮助。

范例：

```
VECTOR3D n={1,1,1};
POINT3D p={0,0,0};
PLANE3D plane;

// 创建平面
PLANE3D_Init(&plane, &p, &n, TRUE);
```

函数原型：

```
float Compute_Point_In_Plane3D(POINT3D_PTR pt, PLANE3D_PTR plane);
```

函数代码：

```
float Compute_Point_In_Plane3D(POINT3D_PTR pt, PLANE3D_PTR plane)
{
// 检测点是在平面上、正半空间还是负半空间中
float hs = plane->n.x*(pt->x - plane->p0.x) +
      plane->n.y*(pt->y - plane->p0.y) +
      plane->n.z*(pt->z - plane->p0.z);
```

```
// 返回检测结果
return(hs);

} // end Compute_Point_in_Plane3D
```

用途：

float Compute_Point_In_Plane3D()是一个很有用的函数。它判断指定点位于哪个半空间中。很多时候，需要判断某样东西位于平面的哪一边，该函数正好提供了这种功能。图 5.21 说明了该函数的逻辑。如果指定的点位于平面上，该函数返回 0.0；如果位于正半空间中，返回一个正数；如果位于负半空间中，返回一个负数。请参考附带光盘中的 DEMOII5_2.CPP|EXE，它允许用户定义一个平面，然后判断指定的点位于平面的哪个半空间中。

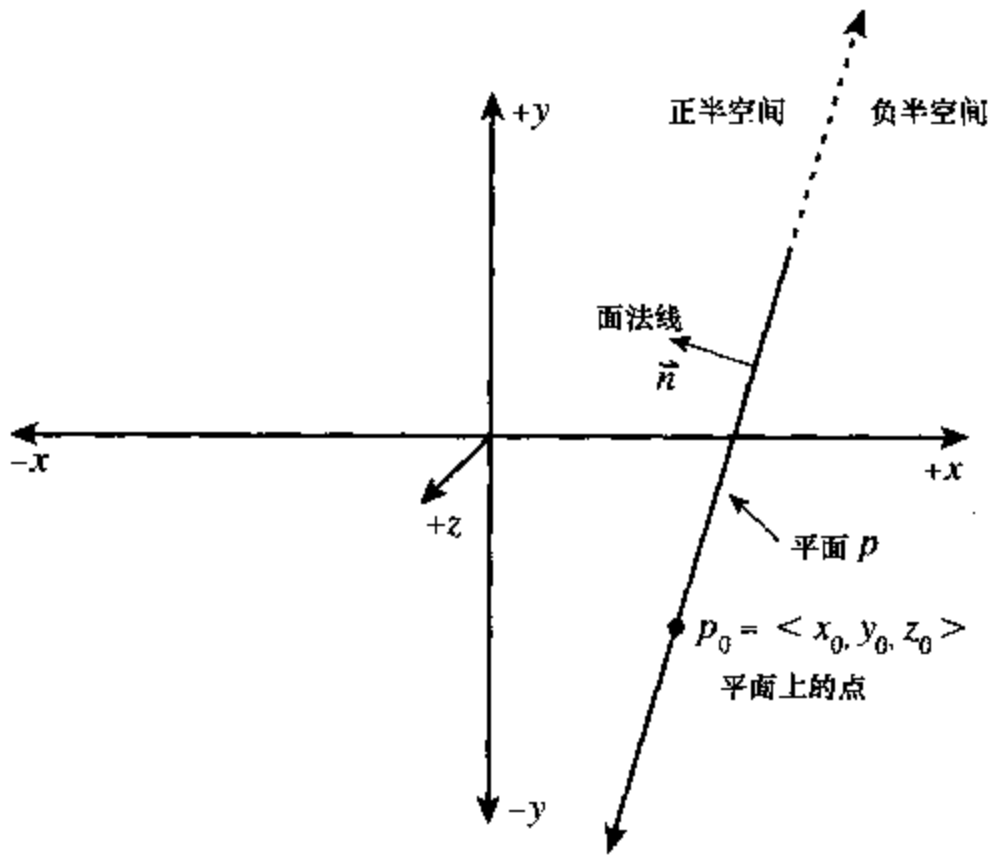


图 5.21　半空间逻辑编码

范例：

```
VECTOR3D n={1,1,1};
POINT3D p={0,0,0};
PLANE3D plane;

// 创建平面
PLANE3D_Init(&plane, &p, &n, TRUE);

// 该点应该在正半空间中
POINT3D p_test = {50,50,50};

// 进行检测
float hs = Compute_Point_In_Plane3D(&p_test, &plane);
```

函数原型：

```
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline, PLANE3D_PTR plane,
                    float *t, POINT3D_PTR pt);
```

函数代码：

```
int Intersect_Parm_Line3D_Plane3D(PARMLINE3D_PTR pline,
                    PLANE3D_PTR plane,
                    float *t, POINT3D_PTR pt)
{
// 这个函数计算参数化直线与平面的交点
// 计算交点时，该函数将参数化直线视为无穷长
// 如果交点在线段 pline 上，则参数 t 的值将位于范围[0,1]内
```

```
// 另外，如果不相交，该函数返回 0，如果交点在线段上，返回 1
// 如果交点不在线段上，2 返回，如果线段位于平面上，则返回 3

// 首先判断线段和平面是否平行
// 如果是，则它们不可能相交，除非线段位于平面上

float plane_dot_line = VECTOR3D_Dot(&pline->v, &plane->n);

if (fabs(plane_dot_line) <= EPSILON_E5)
  {
  // 线段与平面平行
  // 它是否在平面上?
  if (fabs(Compute_Point_In_Plane3D(&pline->p0, plane)) <= EPSILON_E5)
   return(PARM_LINE_INTERSECT_EVERYWHERE);
  else
   return(PARM_LINE_NO_INTERSECT);
  } // end if

// 第 4 章介绍过，可以按下述方式计算交点对应的 t 值
// a*(x0+vx*t) + b*(y0+vy*t) + c*(z0+vz*t) + d =0
// t = -(a*x0 + b*y0 + c*z0 + d)/(a*vx + b*vy + c*vz)
// x0、y0、z0、vx、vy、vz 定义了线段
// d = (-a*xp0 - b*yp0 - c*zp0)， xp0、yp0、zp0 定义了平面上的一个点

*t = -(plane->n.x*pline->p0.x +
     plane->n.y*pline->p0.y +
     plane->n.z*pline->p0.z -
     plane->n.x*plane->p0.x -
     plane->n.y*plane->p0.y -
     plane->n.z*plane->p0.z) / (plane_dot_line);
// 将 t 代入参数化直线方程，以计算 x、y、z
pt->x = pline->p0.x + pline->v.x*(*t);
pt->y = pline->p0.y + pline->v.y*(*t);
pt->z = pline->p0.z + pline->v.z*(*t);

// 检测 t 是否在范围[0, 1]内
if (*t>=0.0 && *t<=1.0)
  return(PARM_LINE_INTERSECT_IN_SEGMENT );
else
  return(PARM_LINE_INTERSECT_OUT_SEGMENT);

} // end Intersect_Parm_Line3D_Plane3D
```

用途：

int Intersec_Parm_Line3D_Plane3D()函数计算一条 3D 参数化直线与一个 3D 平面的交点，将交点处的参数值存储到 *t* 中，并将交点存储到 *pt* 中。然而，在使用这些数据之前，需要测试 Intersec_Parm_Line3D_Plane3D()函数的返回值，以确定是否存在交点。返回值与计算 2D 交点的函数相同，但增加了一个标记，用于指出点位于平面上。下面列出了这些标记：

```
#define PARM_LINE_NO_INTERSECT     0
#define PARM_LINE_INTERSECT_IN_SEGMENT 1
#define PARM_LINE_INTERSECT_OUT_SEGMENT 2
#define PARM_LINE_INTERSECT_EVERYWHERE 3
```

请参考附带光盘中的 DEMOII5_3.CPP|EXE，它允许用户义一个 3D 平面和一条参数化直线，然后计算它们的交点。

范例：

```
POINT3D p1 = {5,5,-5}, p2 = {5,5,5},pt;
PARMLINE3D pl;
float t;

// 创建一条从 p1 到 p2 的参数化直线
// 这条直线与 z 轴平行
Init_Parm_Line3D(&p1, &p2, &pl);
```

```
VECTOR3D n={0,0,1};
POINT3D p={0,0,0};
PLANE3D plane;

// 创建平面
// 该平面与 x-y 平面平行
PLANE3D_Init(&plane, &p, &n, TRUE);

// 计算交点
// 应为(5,5,0)
int intersection_type =
  Intersect_Parm_Line3D_Plane3D(&pl, &plane, &t, &pt);
```

5.7.7 四元数支持函数

必须指出的是，编写四元数函数并不容易！问题在于必须确保它们能正常工作。需要在草稿纸上进行双重检查，因为四元数没有太多的物理意义，无法以图形方式进行检查。然而，最终它们都能正常工作。我只省略了几个矩阵-四元数转换函数和球面线性插值函数。需要时我们将添加，很可能是在介绍相机和动画时。尽管如此，我非常喜欢四元数，因为它使得很多操作都变得非常简单。例如，对于绕任意直线旋转的问题，如果使用四元数来求解将非常容易，而使用欧拉方程则非常复杂。前面介绍过，四元数就是一个超复数，它有 4 个分量，其中一个为实部（通常称为 q0 或 w），另外 3 个为虚部（通常称为 q1、q2、q3）。另外，四元数通常书写为如下形式之一：

$q = q_0 + q_1*i + q_2*j + q_3*k$

或

$q = q_0 + q_v$，其中 $q_v = q_1*i + q_2*j + q_3*k$

因此，我们基本上需要跟踪 4 个浮点数，且应该能够以多种方式访问它们。下面的数据结构 QUAT 使用几个共用体实现了这种功能：

```
typedef struct QUAT_TYP
{
union
  {
  float M[4]; // 按顺序 w、x、y、z 以数组方式存储

  // "向量部分 + 实部"格式
  struct
     {
     float  q0; // 实部
     VECTOR3D qv; // 虚部（xi+yj+zk）
     };
  struct
     {
     float w,x,y,z;
     };
  }; // end union

} QUAT, *QUAT_PTR;
```

使用这个数据结构，可以按数组、实数和向量或 4 个惟一标识符来访问四元数。这使得编写各种算法更简单！下面介绍四元数函数。

函数原型：

```
void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v, float theta);

void VECTOR4D_Theta_To_QUAT(QUAT_PTR q, VECTOR4D_PTR v, float theta);
```

函数代码：

```
void VECTOR3D_Theta_To_QUAT(QUAT_PTR q, VECTOR3D_PTR v, float theta)
{
// 使用一个 3D 方向向量和一个角度来初始化一个四元数
```

```
// 方向向量必须是单位向量，角度的单位为弧度

float theta_div_2 = (0.5)*theta; // 计算 theta/2

// 计算四元数，这在第 4 章介绍过
// 预先计算以节省时间
float sinf_theta = sinf(theta_div_2);

q->x = sinf_theta * v->x;
q->y = sinf_theta * v->y;
q->z = sinf_theta * v->z;
q->w = cosf( theta_div_2 );

} // end VECTOR3D_Theta_To_QUAT
```

用途：

void VECTOR*D_Theta_To_QUAT() 根据方向向量 v 和角度 theta 创建一个旋转四元数。图 5.22 说明了其中的关系。这个函数主要用于创建对点进行旋转的四元数。注意，方向向量 v 必须为单位向量。另外，4D 向量版本将丢弃 *w* 分量。

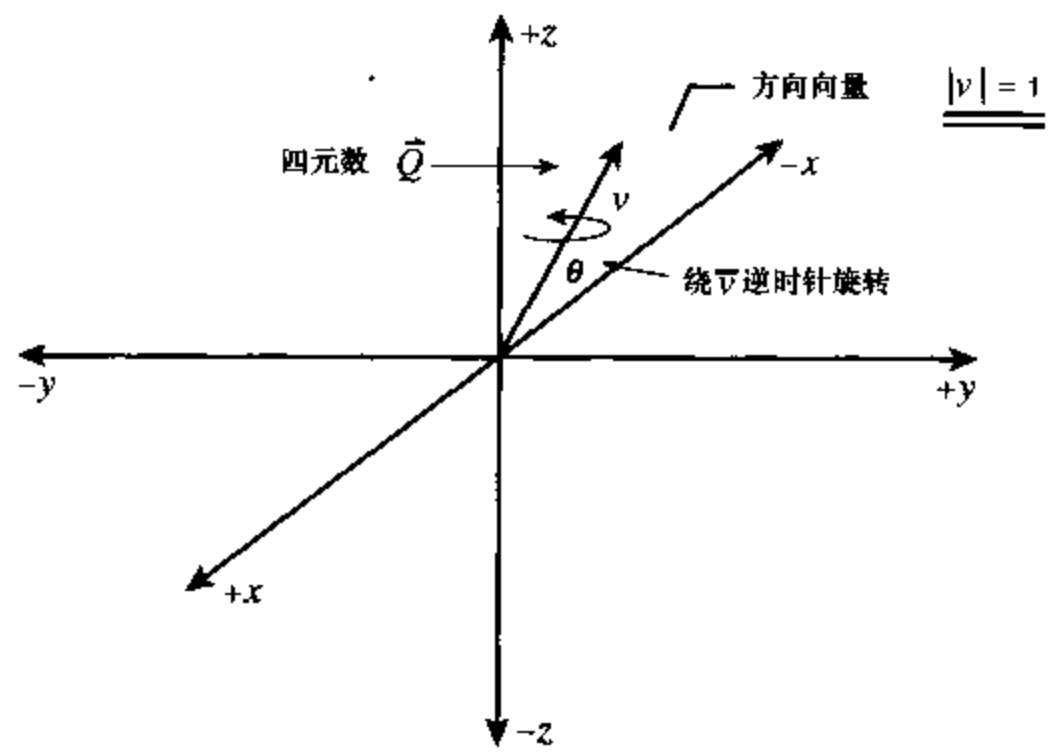


图 5.22　创建旋转四元数

范例：

```
// 创建要绕其旋转的向量
// 这里为卦限 1 对应的立方体的对角线
VECTOR3D v={1,1,1};
QUAT qr;

// 对向量 v 进行归一化
VECTOR3D_Normalize(&v);

float theta = DEG_TO_RAD(100); // 100 度

// 创建一个绕向量 v 旋转 100 度的四元数
VECTOR3D_Theta_To_QUAT(&q, &v,theta);
```

函数原型：

```
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z, float
theta_y, float theta_x);
```

函数代码：

```
void EulerZYX_To_QUAT(QUAT_PTR q, float theta_z, float theta_y, float theta_x)
{
// 这个函数根据绕 x、y、z 旋转的角度，创建一个的 zyx 顺序进行旋转对应的四元数
// 注意，还有 11 个根据旋转角度创建四元数的函数
```

```
// 本书后面可能编写该函数的通用版本

// 预先计算一些值
float cos_z_2 = 0.5*cosf(theta_z);
float cos_y_2 = 0.5*cosf(theta_y);
float cos_x_2 = 0.5*cosf(theta_x);

float sin_z_2 = 0.5*sinf(theta_z);
float sin_y_2 = 0.5*sinf(theta_y);
float sin_x_2 = 0.5*sinf(theta_x);

// 计算四元数
q->w = cos_z_2*cos_y_2*cos_x_2 + sin_z_2*sin_y_2*sin_x_2;
q->x = cos_z_2*cos_y_2*sin_x_2 - sin_z_2*sin_y_2*cos_x_2;
q->y = cos_z_2*sin_y_2*cos_x_2 + sin_z_2*cos_y_2*sin_x_2;
q->z = sin_z_2*cos_y_2*cos_x_2 - cos_z_2*sin_y_2*sin_x_2;

} // EulerZYX_To_QUAT
```

用途：

void EulerZYX_To_QUAT()根据绕 z、y、x 旋转的欧拉角创建一个旋转四元数。它是一种基本的相机转换。当然，旋转顺序共有 6 种（3 的阶乘），但这种顺序是最常见的。使用这个函数可将欧拉旋转角转换为四元数。

范例：

```
QUAT qzyx;

// 创建旋转角度
float theta_x = DEG_TO_RAD(20);
float theta_y = DEG_TO_RAd(30);
float theta_z = DEG_TO_RAD(45);

// 创建旋转四元数
EulerZYX_To_QUAT(&qzyx,theta_z,theta_y,theta_x);
```

函数原型：

```
void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v, float *theta);
```

函数代码：

```
void QUAT_To_VECTOR3D_Theta(QUAT_PTR q, VECTOR3D_PTR v, float *theta)
{
// 这个函数将一个单位四元数转换为一个单位方向向量和一个绕该向量旋转的角度

// 提取角度
*theta = acosf(q->w);

// 预先计算以节省时间
float sinf_theta_inv = 1.0/sinf(*theta);

// 计算向量
v->x  = q->x*sinf_theta_inv;
v->y  = q->y*sinf_theta_inv;
v->z  = q->z*sinf_theta_inv;

// 将角度乘以 2
*theta*=2;

} // end QUAT_To_VECTOR3D_Theta
```

用途：

void QUAT_ToVECTOR3D_Theta()函数将一个单位旋转四元数转换为一个单位 3D 向量和与一个绕该向量旋转的角度 theta。这个函数是 VECTOR*D_Theta_To_QUAT()函数的反函数。

范例：

```
QUAT q;
// 假设 q 是一个单位四元数

// 用于存储向量和角度的变量
float theta;
VECTOR3D v;

// 将四元数转换为向量和角度
QUAT_To_VECTOR3D_Theta(&q, &v, &theta);
```

函数原型：

```
void QUAT_Add(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qsum);
```

用途：

void QUAT_Add() 函数将两个四元数 q1 和 q2 相加，并将结果存储到 qsum 中。

范例：

```
QUAT q1 = {1,2,3,4}, q2 = {5,6,7,8}, qsum;

// 将它们相加
QUAT_Add(&q1, &q2, &qsum);
```

函数原型：

```
void QUAT_Sub(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qdiff);
```

用途：

void QUAT_Sub() 函数将两个四元数 q1 和 q2 相减，并将结果存储到 qdiff。

范例：

```
QUAT q1 = {1,2,3,4}, q2 = {5,6,7,8}, qdiff;

// 将它们相减
QUAT_Sub(&q1, &q2, &qdiff);
```

函数原型：

```
void QUAT_Conjugate(QUAT_PTR q, QUAT_PTR qconj);
```

用途：

void QUAT_Conjugate() 函数计算四元数 q 的共轭，并将结果存储到 qconj 中。

范例：

```
QUAT q = {1,2,3,4}, qconj;

// 计算共轭
QUAT_Conjugate(&q, &qconj);
```

函数原型：

```
void QUAT_Scale(QUAT_PTR q, float scale, QUAT_PTR qs);
```

用途：

void QUAT_Scale() 函数根据缩放因子 scale 对四元数 q 进行缩放，并将结果存储到 qs 中。

范例：

```
QUAT q = {1,2,3,4}, qs;

// 将 q 放大 2 倍
QUAT_Scale(&q, 2, &qs);
```

函数原型：

```
void QUAT_Scale(QUAT_PTR q, float scale);
```

用途：

void QUAT_Scale() 函数根据缩放因子 scale 直接缩放四元数 q，即修改 q。

范例：

```
QUAT q = {1,2,3,4};

// 将 q 放大 2 倍
QUAT_Scale(&q, 2);
```

函数原型：

```
float QUAT_Norm(QUAT_PTR q);
```

用途：

float QUAT_Norm（QUAT_PTR q）函数返回四元数 q 的范数，即长度。

范例：

```
QUAT q = {1,2,3,4};

// 计算 q 的长度
float qnorm = QUAT_Norm(&q);
```

函数原型：

```
float QUAT_Norm2(QUAT_PTR q);
```

用途：

float QUAT_Norm2（QUAT_PTR q）函数返回四元数 q 的范数平方，即长度平方。该函数很有用，因为很多时候我们都需要计算四元数的范数平方。与调用 QUAT_Norm()，然后计算返回值的平方相比，调用 QUAT_Norm2() 可节省一次 sqrt() 函数调用和一次乘法。

范例：

```
QUAT q = {1,2,3,4};

// 计算范数平方
float qnorm2 = QUAT_Norm2(&q);
```

函数原型：

```
void QUAT_Normalize(QUAT_PTR q, QUAT_PTR qn);
```

用途：

void QUAT_Normalize() 函数将四元数 q 归一化，并将结果存储到 qn 中。

提示：别忘了，所有旋转四元数都必须为单位四元数。

范例：

```
QUAT q = {1,2,3,4}, qn;

// 对 q 进行归一化
QUAT_Normalize(&q, &qn);
```

函数原型：

```
void QUAT_Normalize(QUAT_PTR q);
```

用途：

void QUAT_Normalize() 直接对四元数 q 进行归一化，即修改 q 的值。

范例：

```
QUAT q = {1,2,3,4};

// 就地对 q 进行归一化
QUAT_Normalize(&q);
```

函数原型：

```
void QUAT_Unit_Inverse(QUAT_PTR q, QUAT_PTR qi);
```

用途：

void QUAT_Unit_Inverse() 函数计算四元数 q 的逆，并将结果存储到到 qi 中。然而，q 必须为单位四元数，因为该函数基于单位四元数的逆就是其共轭这一原理来求逆。

范例：

```
QUAT q = {1,2,3,4}, qi;

// 首先归一化 q
QUAT_Normalize(&q);

// 现在计算其逆
QUAT_Unit_Inverse(&q, &qi);
```

函数原型：

```
void QUAT_Unit_Inverse(QUAT_PTR q);
```

用途：

void QUAT_Unit_Inverse() 函数计算四元数 q 的逆，并直接修改 q。然而，q 必须为单位四元数，因为该函数根据单位四元数的逆就是其共轭这一原理来求逆。

范例：

```
QUAT q = {1,2,3,4};

// 首先归一化 q
QUAT_Normalize(&q);

// 现在计算其逆
QUAT_Unit_Inverse(&q);
```

函数原型：

```
void QUAT_Inverse(QUAT_PTR q, QUAT_PTR qi);
```

用途：

void QUAT_Inverse() 函数计算非单位四元数 q 的逆，并将结果存储到 qi 中。

范例：

```
QUAT q = {1,2,3,4}, qi;

// 计算逆
QUAT_Inverse(&q, &qi);
```

函数原型：

```
void QUAT_Inverse(QUAT_PTR q);
```

用途：

void QUAT_Inverse() 函数直接计算非单位四元数 q 的逆，即修改 q 的值。

范例：

```
QUAT q = {1,2,3,4};

// 计算逆
QUAT_Inverse(&q);
```

函数原型：

```
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod);
```

函数代码：

```
void QUAT_Mul(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR qprod)
{
// 这个函数将两个四元数相乘

// 这是一种蛮干方法
//qprod->w = q1->w*q2->w - q1->x*q2->x - q1->y*q2->y - q1->z*q2->z;
//qprod->x = q1->w*q2->x + q1->x*q2->w + q1->y*q2->z - q1->z*q2->y;
//qprod->y = q1->w*q2->y - q1->x*q2->z + q1->y*q2->w - q1->z*q2->x;
```

```
//qprod->z = q1->w*q2->z + q1->x*q2->y - q1->y*q2->x + q1->z*q2->w;

// 下述方法先计算共用因子，以减少乘法运算次数

float prd_0 = (q1->z - q1->y) * (q2->y - q2->z);
float prd_1 = (q1->w + q1->x) * (q2->w + q2->x);
float prd_2 = (q1->w - q1->x) * (q2->y + q2->z);
float prd_3 = (q1->y + q1->z) * (q2->w - q2->x);
float prd_4 = (q1->z - q1->x) * (q2->x - q2->y);
float prd_5 = (q1->z + q1->x) * (q2->x + q2->y);
float prd_6 = (q1->w + q1->y) * (q2->w - q2->z);
float prd_7 = (q1->w - q1->y) * (q2->w + q2->z);

float prd_8 = prd_5 + prd_6 + prd_7;
float prd_9 = 0.5 * (prd_4 + prd_8);

// 现在使用临时乘积计算最后结果
qprod->w = prd_0 + prd_9 - prd_5;
qprod->x = prd_1 + prd_9 - prd_8;
qprod->y = prd_2 + prd_9 - prd_7;
qprod->z = prd_3 + prd_9 - prd_6;

} // end QUAT_Mul
```

最开始我使用直接方法，根据乘法定义将四元数相乘（共 16 次乘法，12 次加法）；然后使用一些代数公式，将计算简化为 9 次乘法和 27 次加法。通常，您可能认为后一种方法更好，但在浮点处理器上可能并非如此。

用途：

void QUAT_Mul() 函数用于将四元数相乘，并将结果存储到 qprod 中。

范例：

```
QUAT q1={1,2,3,4}, q2={5,6,7,8}, qprod;

// multiply q1*q2
QUAT_Mul(&q1, &q2, qprod);
```

警告：乘积 q1*q2 并不等于 q2*q1，除非 q1 或 q2 为乘法单位数。因此，一般来说，四元数乘法不满足交换律。

函数原型：

```
void QUAT_Triple_Product(QUAT_PTR q1, QUAT_PTR q2, QUAT_PTR q3, QUAT_PTR qprod);
```

用途：

void QUAT_Triple_Product() 函数将 3 个四元数相乘，并将结果存储到 qprod 中。该函数对点旋转很有用，因为变换（**q***）*（v）*（**q**）和（**q**）*（v）*（**q***）是用于旋转向量或点的三重乘积。

范例：

```
// 将点(5,0,0)绕 z 轴旋转 45 度

// 第 1 步：创建旋转四元数
VECTOR3D vz = {0,0,1};

QUAT qr, // 用于存储旋转四元数
   qrc; // 用于存储其共轭

// 创建旋转四元数
VECTOR3D_Theta_To_QUAT(&qr, &vz, DEG_TO_RAD(45));

// 计算其共轭
QUAT_Conjugate(&qr, &qrc);

// 现在创建一个表示点的四元数，将 q0 设置为 0
// 将 x、y、z 设置与前述点相应的坐标
```

```
QUAT qp={0,5,0,0};

// 现在执行旋转，将点 p 绕 z 轴旋转 45 度
// 当然，旋转轴可以是任何轴
QUAT_Triple_Product(&qr, &qp, &qrc, &qprod);

// 结果中 q0 仍然为 0
// 我们只需从四元数中提取 x、y 和 z，便可得到旋转后的点
```

函数原型：

```
void QUAT_Print(QUAT_PTR q, char *name);
```

用途：

void QUAT_Print() 函数使用指定名称将一个四元数输出到错误文件中。

范例：

```
// 将错误文件设置为屏幕
Open_Error_File(" ", stdout);

QUAT q={1,2,3,4};

QUAT_Print(&q);

// 关闭错误文件
Close_Error_File();
```

附带光盘中的 DEMOII5_4.CPP|EXE 演示了一些四元数函数。它让用户能够输入两个四元数和 3D 空间中的一个点，然后对它们执行各种运算并查看结果。

5.7.8　定点数支持函数

虽然前一章没有介绍定点数，但我认在数学引擎中添加一些操作定点数的函数是值得的。有关这个主题的详细介绍，请参阅《Windows 游戏编程大师技巧》。另外，在奔腾以上的处理器上，浮点数单元的速度与整数单元一样快，甚至更快，因此不再需要使用定点数来执行 3D 数学运算。然而，很多时候，同时使用浮点数单元和整数单元可以编写出真正优化的算法，因此了解如何表示和操作定点数仍是件好事。

一般来说，定点数是一种较早的技巧，用于在较慢的处理器使用整数来执行乘法和除法，但有小数精度。其技巧是将一个“固定小数点”放在某个数的二进制表示中，从而将该数划分为整数部分和小数部分。在数学引擎中，我将使用《Windows 游戏编程大师技巧》中采用的 16.16 格式，即使用 16 位表示整数部分，使用 16 位表示小数部分。这意味着它可以表示的数值为+/-32768，小数精度为 $1/2^{16}$。

现在的问题是，如何表示定点数？答案是使用标准的 int 或 long 数据类型（如果需要 64 位，应使用 long）。因此，T3DLIB4.H 中定义了如下类型：

```
typedef int FIXP16;
typedef int *FIXP16_PTR;
```

在 T3DLIB1.H 中还定义了下述常量：

```
// 与定点数运算相关的常量
#define FIXP16_SHIFT   16
#define FIXP16_MAG     65536
#define FIXP16_DP_MASK  0x0000ffff
#define FIXP16_WP_MASK  0xffff0000
#define FIXP16_ROUND_UP 0x00008000
```

这些常量用于简化定点数运算。

1. 使用整数创建定点数

要使用整数创建定点数，只需将该整数左移 FIXP16_SHIFT 位，将其放到定点数的整数部分中。但记住不要让定点数溢出。下面是一个范例：

```
FIXP16 fp1 = (100 << FIXP16);
```

当然，头文件中有一个名为 INT_TO_FIXP16() 的宏用于执行这种操作。

2. 使用浮点数创建定点数

使用浮点数来创建定点数要稍微复杂些，因为整数的二进制表示与浮点数的二进制表示有很大的不同。因此，需要对浮点数进行缩放，缩放比例与对整数进行移位的程度相同。将 int 左移 16 位，将其转换为定点数时，相当于将其乘以 $2^{16} = 65536$。因此，如果将要转换的浮点数乘以 65536.0，然后将它转换为 int，将可以解决这个问题。下面是一个范例：

```
FIXP16 fp1 = (int)(100.5*65536.0);
```

注意，这里没有执行移位操作，因为乘法的效果与移位相同。当然，在头文件中也有一个名为 FLOAT_TO_FIXP16() 的宏，用于执行这种功能。然而，您可能想对该数进行取整，因为将它转换为 int 时将截去小数部分。因此，在执行转换之前，将它加上 0.5，以免丢失数据，最终的宏将执行该操作，它与下面类似：

```
FIXP16 fp1 = (int)(100.5*65536.0+0.5);
```

3. 转换回浮点数

要将定点数转换回浮点数，首先需要提取定点数的整数部分，这并不困难，因为它就是定点数的前 16 位，且按正常的整数格式存储。然而，获得小数部分需要一些技巧。需要将小数部分除以原来缩放的幅度 65536。然而，如果不想分别提取整数部分和小数部分，可以使用下列变换将定点数转换为浮点数：

```
float ((float)fp)/65536.0;
```

上述代码位于宏 FIXP16_TO_FLOAT() 中。

下面介绍如何对两个定点数执行基本运算。

（1）加法/减法

要将两个用定点数格式表示的数相加或相减，只需要按正常方法执行加法或减法。例如：

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);

FIXP16 fpsum = fp1 + fp2;
FIXP16 fpdiff = fp1 - fp2;
```

为更清晰地说明这一点，再来看一个范例：

```
int x=50, y=23;

// 将 x、y 转换为 16.16 格式的定点数
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
FIXP16 fsum;

// 现在将它们相加
fsum = fpx + fpy;
```

上述代码可简化为：

```
fsum = x*65536 + y*65536 = (x+y)*65536;
```

结果表明，的确执行了加法（x+y），且缩放了正确的比例——65536。减法的证明过程类似。

（2）乘法

定点数的乘法有些棘手，其原因是每个数都被乘以了一个因子。使用 16.16 格式时，每个定点数都被乘以了 65536。因此，如果将两个定点数相乘，需要确保结果的缩放倍数为 65536，而不是 65536^2！下面的范例说明了这种问题：

```
// 要转换为定点数的数字
int x=50, y=23;
```

```
// 将 x、y 转换为 16.16 格式的定点数
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

现在，当我们计算（fpx*fpy）时，问题出现了。如果查看数值结果，将看到如下内容：

```
fpx*fpy = (x*65536) * (y*65536) = (x*y)*(65536*65536);
```

发现问题了吗？结果的缩放倍数为 65536^2，而不是 65536。其解决方法是将结果缩放正确的倍数。然而，真正的问题在于，将两个定点数相乘（然后再缩放到正确的倍数）会导致 32 位数溢出。因此，要么使用 64 位数（稍后介绍），要么先缩小乘数，然后再执行乘法，使结果的缩放倍数为 65536。下面是一种完成这项任务的方法：

```
= (fpx/256)*(fpy/256)
```

替换为 x、y，结果如下：

```
= ((x*65536)/256) * ((y*65536)/256)
= (x*256)*(y*256) = (x*y)*65536
```

这是正确的，但这种方法的问题在于，在缩小期间，乘数和被乘数的精度都降低了 8 位，这是不能接受的，因为定点数的主旨就是要保留小数点精度不变。然而，可以使用汇编语言用 64 位数来处理这种问题。

（3）除法

除法与乘法有相同的问题，但正好相反。它的问题是，将一个定点数除以另一个定点数时，小数部分将丢失，即结果的缩放倍数为 1.0，而不是 65536。下面的范例演示了这一点：

```
// 要将其转换为定点数的数字
int x=50, y=23;

// 将 x 和 y 转换为 16.16 格式的定点数
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

现在执行除法运算：

```
fpx/fpy = (x*65536) / (y*65536) = (x/y);
```

这显然很糟糕。同样，有两种方法可以解决这种问题。可以在汇编语言中使用 64 位除法，也可以预先缩放分子和分母，使除法运算的结果被缩放 65536 倍。然而，如果使用类型 int，同样会受到 32 位数的限制。放大分子意味着降低整数部分的精度，而缩小分母意味着降低小数部分的精度，但我们别无选择。因此，放大分子 256 倍（8 位），并缩小分母 256 倍（8 位），除法结果的缩放倍数将为正确的 65536。然而，在缩放转换期间，分子从 16.16 格式变成了 8.16 格式，而分母从 16.16 格式变成了 16.8 格式，这意味着如果要处理带符号的数值，则分子不能大于 255，分母不能小于 $1/2^8 \approx 0.0039$。

下面是修正后的范例：

```
// 要转换为定点数的数字
int x=50, y=23;

// 将 x、y 转换为 16.16 格式的定点数
FIXP16 fpx = x*65536;
FIXP16 fpy = y*65536;
```

现在来除以缩放因子，看看发生的情况：

```
= (x*256*65536) / (y*65536/256)
= (x*256*256/y)*65536/65536
= (x/y)*(256*256) = (x/y)*65536
```

提示：这些范例表明，并没有规定所有定点数都必须采用 16.16 格式；有些数可以为 16.16 格式，有些可以为 24.8，也有些可以为 0.32 格式。关键在于在执行运算期间要尽可能保持精度，并防止溢出。

下面介绍具体的函数及其内部的汇编语言代码。

函数原型：

```
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2);
```

函数代码：

```
FIXP16 FIXP16_MUL(FIXP16 fp1, FIXP16 fp2)
{
// 这个函数计算 fp1*fp2
// 它使用 64 位数，因此不会降低精度

FIXP16 fp_prod; // 用于存储结果

_asm {
   mov eax, fp1   // 将 fp1 移到 eax 中
   imul fp2     // 计算 fp1*fp2
   shrd eax, edx, 16 // 结果为 32.32 格式
            // 位于 edx:eax 中
            // 将其移位到 eax 中，变成 16.16 格式
   // 结果位于 eax 中
   } // end asm

} // end FIXP16_MUL
```

这些汇编语言非常直观；惟一需要特别注意的是 Intel 公司的 32 位处理器能够支持 64 位运算！

用途：

FIXP16 FIX16_MUL() 函数使用 64 位数将两个定点数相乘，并返回结果。由于它使用了 64 位数，所以不会降低精度。

范例：

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);
// 执行乘法运算
FIXP16 fpprod = FIXP16_Mul(fp1,fp2);
```

函数原型：

```
FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2);
```

函数代码：

```
FIXP16 FIXP16_DIV(FIXP16 fp1, FIXP16 fp2)
{
// 这个函数使用 64 位数计算 fp1/fp2，以免降低精度

_asm {
   mov eax, fp1   // 将被除数移到 eax 中
   cdq         // 带符号扩展到 edx:eax 中
   shld edx, eax, 16 // 对 edx 进行移位
   sal eax, 16    // 对 eax 进行移位
   idiv fp2     // 执行除法运算
   // 结果位于 eax 中
   } // end asm

} // end FIXP16_DIV
```

除法算法有点复杂。首先被除数必须带符号扩展为 64 位，且 64 位数的位移操作有点复杂，因为 64 位双位移指令不会对源寄存器 eax 执行位移——它保持不变。因此，在最初的 64 位移位后，需要一个额外的移位。关于这些 64 位移位指令只完成了一半工作的更多信息，请参考 Intel 公司的汇编语言手册。

用途：

FIXP16 FIXP16_DIV() 函数执行除法 fp1/fp2，并返回结果。注意，由于它使用了 64 位数，所以不会降低精度。

范例：

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16 fp2 = FLOAT_TO_FIX(20.7);

// 执行除法
FIXP16 fpdiv = FIXP16_Mul(fp1,fp2);
```

注意：虽然定点数用于解决浮点数的效率问题，但正如我们看到的，定点数乘法和除法操作都涉及到大量的指令。定点数加法和减法的速度通常与浮点数相同；而浮点数乘法和除法，通常与定点数的速度一样快，甚至更快。

函数原型：

```
void FIXP16_Print(FIXP16 fp);
```

用途：

void FIXP16_Print() 函数以浮点数格式打印定点数。

范例：

```
FIXP16 fp1 = FLOAT_TO_FIX(10.5);
FIXP16_Print(fp1);
```

最后，请参考附带光盘中的 DEMOII5_5.CPP|EXE，它让用户能够输入两个浮点数，将它们转换为定点数，对它们执行各种运算，然后打印出结果，让您能够看到精度。

5.7.9　方程求解支持函数

接下来要介绍的两个函数很有用，它们让您能够求解 A*X=B 形式的方程组。您只需要提供系数矩阵 A 和常量矩阵 B，如果方程组有解，函数将计算正确的解。例如，给定一个 3×3 的方程组：

```
3*x + 2*y - 5*z = 6
  x - 3*y + 7*z = -4
5*x + 9*y - 2*z = 5
```

我们得到：

$$A = \begin{bmatrix} 3 & 2 & -5 \\ 1 & -3 & 7 \\ 5 & 9 & -2 \end{bmatrix}$$

$$X = [x\ y\ z]^t$$

$$B = [6\ -4\ 6]^t$$

然后可以使用变换 $X=A^{-1}*B$，这就是方程求解函数所做的。

函数原型：

```
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X, MATRIX1X2_PTR B);
```

函数代码：

```
int Solve_2X2_System(MATRIX2X2_PTR A, MATRIX1X2_PTR X, MATRIX1X2_PTR B)
{
// 求解方程组 AX=B
// 使用克莱姆法则和行列式计算 X=A(-1)*B

// 第 1 步：计算 A 的行列式值
float det_A = Mat_Det_2X2(A);

// 检测 det_A 是否为零，如果是，则方程组无解
if (fabs(det_A) < EPSILON_E5)
  return(0);

// 第 2 步：分别将矩阵 A 中的 X、Y 列替换为矩阵 B，得到分子矩阵
// 以计算 x 和 y 的解
```

```
MATRIX2X2 work_mat; // 工作矩阵

// 求解 x ////////////////////

// 将 A 复制到工作矩阵中
MAT_COPY_2X2(A, &work_mat);

// 替换 X 列
MAT_COLUMN_SWAP_2X2(&work_mat, 0, B);

// 计算替换后的行列式值
float det_ABx = Mat_Det_2X2(&work_mat);
// 计算 X 的解
X->M00 = det_ABx/det_A;

// 求解 Y ////////////////////

// 将 A 复制到工作矩阵中
MAT_COPY_2X2(A, &work_mat);

// 替换 Y 列
MAT_COLUMN_SWAP_2X2(&work_mat, 1, B);

// 计算替换后的行列式值
float det_ABy = Mat_Det_2X2(&work_mat);

// 计算 Y 的解
X->M01 = det_ABy/det_A;

// 成功返回
return(1);

} // end Solve_2X2_System
```

用途：

int Solve_2×2_System()函数用于求解方程组 A*X=B，其中 A 为 2×2，X 为 1×2，B 为 1×2 矩阵。如果有解，将其存储在 B 中，且函数返回 1；否则函数返回 0，且 B 未定义。另外，请参考函数代码，查看函数的内部结构。该函数使用克莱姆法则来解方程组。Solve_2×2_System()是诸多矩阵函数中的典范，请请仔细分析该函数。此外，请参考附带光盘中的 DEMOII5_6.CPP|EXE，它让用户能够输入一个方程组，然后求解该方程组。

范例：

DEMOII5_6.CPP 让用户输入系数矩阵 A 和常量矩阵 B，然后使用克莱姆法则求解方程组 A*X=B。下面是 main()函数的代码：

```
void main()
{
MATRIX2X2 mA;
MATRIX1X2 mB;
MATRIX1X2 mX;

// 打开错误系统，以便用户能够看到输出
// 最后一个参数为"stdout"，这告诉错误系统，不将错误信息写入文本文件
// 而是直接输出到屏幕上
Open_Error_File(" ", stdout);

// 读取矩阵 A
printf("\nEnter the values for the matrix A (2x2)");
printf("\nin row major form m00, m01, m10, m11?");
scanf("%f, %f, %f, %f", &mA.M00, &mA.M01, &mA.M10, &mA.M11);

printf("\nEnter the values for matrix B (2x1)");
printf("\nin column major form m00, m10?");
scanf("%f, %f", &mB.M00, &mB.M01);
```

```
// 现在求解方程组
if (Solve_2X2_System(&mA, &mX, &mB))
  {
  // 打印结果
  VECTOR2D_Print((VECTOR2D_PTR)&mX, "Solution matrix mX");
  } // end if
else
  printf("\nNo Solution!");

// 关闭错误系统
Close_Error_File();

} // end main
```

函数原型：

```
int Solve_3X3_System(MATRIX3X3_PTR A, MATRIX1X3_PTR X, MATRIX1X3_PTR B);
```

用途：

int Solve_3×3_System()函数用于求解方程组 A*X=B，其中 A 为 3×3，X 为 1×3，B 为 1×3 矩阵。如果方程组有解，将求得的解存储到 B 中，且返回 1；否则返回 0，且 B 未定义。

范例：

```
MATRIX3X3 mA = {1,2,9, 4,-3,6, 1,0,5};
MATRIX1X3 mB = {1,2,3};
MATRIX1X3 mX;

// 求解方程组
if(Solve_3X3_System(&mA, &mX, &mB))
  {
  // 打印结果
  VECTOR3D_Print((VECTOR3D_PTR)&mX, "Solution matrix mX");
  } // end if
else
  printf("\nNo Solution!");
```

5.8　浮点单元运算初步

80387 和奔腾的内部协处理器（FPU）可能是 PC 处理套件中最为神秘的部件。说实话，除 Intel 指令集文档和两三本难以找到的图书外，几乎没有其他关于 FPU 编程的优秀参考资料。然而，现在情况发生了变化。

注意：当前，新一代奔腾处理器可能有多个 FPU，所以我将浮点数编程和浮点数协处理器统称为 FPU。

这里对 FPU 的介绍至少将达到这样的程度，让读者理解其原理以及如何使用其指令集对其进行编程。当然，您编译使用浮点数运算的代码时，C/C++编译器将生成 FPU 指令。然而，在很多情况下，这些代码可以使用汇编语言和 64 位指令进行优化，就像前面使用整数单元执行 64 位定点数乘法和除法运算那样。

然而，在支持 MMX 的奔腾处理器上，FPU 寄存器同时被用作 MMX 寄存器，这意味着要混合使用 MMX 代码和 FPU 代码，必须执行大量的状态切换。这里不打算介绍 MMX 编程，因为它是一个附加指令集，用于执行 SIMD（single instruction multiple data，单指令多数据）处理，且本身就是一个主题，更适合在关于优化的书籍中介绍。如果读者对它感兴趣，可从 Intel 网站的开发人员部分中获得一本很好的图书：MMX *Programming Guide*。

那么，FPU 是如何工作的呢？首先，从编程的角度来说，不管 FPU 有一个还是多个，是内部的还是外部的（高性能 FPU），指令集都几乎是相同的；至少对于我们要学习的最常用运算来说是这样的。

警告：FPU 编程必须使用汇编语言来进行。但我们将使用内联汇编器，因此即使读者不是汇编语言专

家，学习 FPU 编程也非常容易，因为它像一台有堆栈的计算机。

5.8.1 FPU 体系结构

FPU 是一台基于堆栈的机器，它只有一个用途：帮助 CPU 核心执行浮点数计算。图 5.23 是 FPU 与 CPU 核心关系的抽象模型。基本上，CPU 将所有的 FPU 指令发送给 FPU，在那里执行计算；因此，FPU 指令不会直接影响 CPU 或其内部寄存器。在很多时候，CPU 的整数单元和 FPU 不能同时工作；但在大多数情况下，CPU 和 FPU 可以并行工作，尤其是像奔腾那样采用 U-V 管道（pipe）体系结构的处理器。在奔腾以上的处理器上，事情变得稍微复杂些，所以您可能要了解关于奔腾 I、II、III、Pro 和所有其他型号的规则，因为这里不打算介绍这些规则！

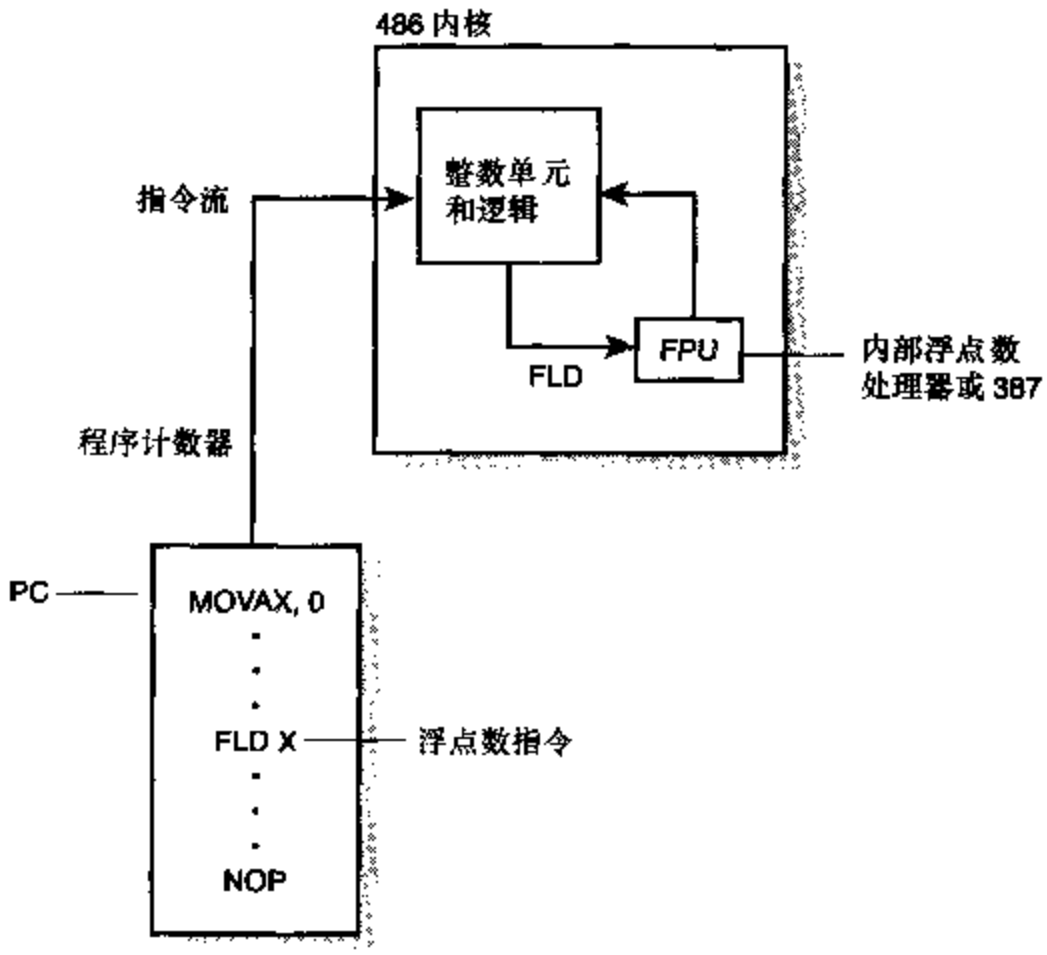


图 5.23 浮点数处理器执行所有浮点数运算

FPU 能够执行很多常见的数学运算，例加法、减法、乘法和除法；还能够处理对数、超越数、数值转换等。另外，FPU 执行很多运算时，速度比整数单元更快，精度更高。虽然执行纯整数运算时，FPU 的速度比整数单元慢，但由于精度更高，因此有时候使用 FPU 也是值得的，尤其是在 3D 图形学计算中。

FPU 编程实际上非常简单。它有自己的指令集，稍后将介绍。基本上，要执行 FPU 指令，只需将其与其他汇编语言指令一起放到代码中即可。这提出了一个小问题，即大多数 FPU 编程都必须使用汇编语言（外部汇编函数或内联汇编函数，作者更喜欢使用后者）来完成。然而，当我编写直接使用 FPU 的函数时，通常全部使用内联汇编语言。当然我只是对于极少数函数这样做，这些函数是必须以非常快的速度运行，且代码不超过 500 行的高级函数。

注意：将编译器设置成进行高度优化和使用内联 FPU 指令时，它将生成已经优化得很好的 FPU 代码。

5.8.2 FPU 堆栈

下面介绍 FPU 的内部堆栈，因为它是一切运算的基础。基本上，FPU 中有 8 个栈元素和一个状态字，如图 5.24 所示。每个栈元素宽 80 位，其中 64 位用于小数部分（decimal part），15 位用于指数，1 位用作符号位。

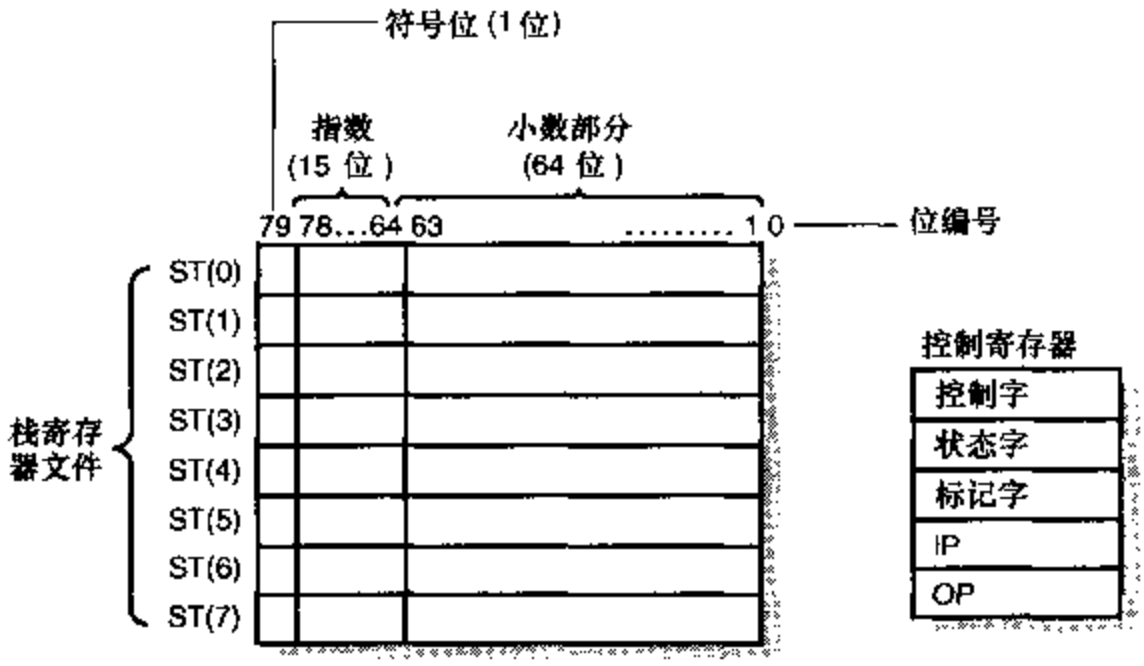


图 5.24　FPU 核心的体系结构

然而，每个栈元素可以存储不同的数据类型，如图 5.25 所示，其中有短实数、长实数和 10 字节实数。栈元素用于存储输入值和返回输出值；因此，从某种意义上说，它们是 FPU 的工作寄存器。可以将它们视为一个寄存器文件或堆栈。

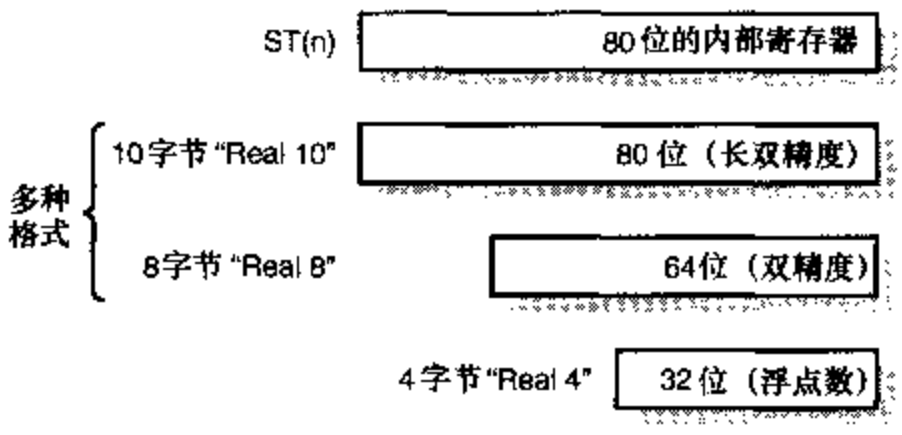


图 5.25　FPU 数据格式

我们将使用句法 ST(*n*)来引用堆栈，其中 *n* 为栈元素编号，如图 5.26 所示。

下面是一些范例：

- ST 表示栈顶（TOS）。
- ST(0)也表示栈顶。
- ST(1)表示堆栈中的第二个元素。
- ST(7)表示堆栈中的最后一个元素（第 8 个元素）。

进行 FPU 编程时，很多指令都接收 0、1 或 2 个操作数。这意味着有时候操作数是隐含的。另外，操作数可以为常量、内存变量或栈元素，所以在指定要对其进行运算的数据时有一定的灵活性。还有，很多指令都将结果存储在栈顶——ST(0)。如果希望某些数据在执行运算后仍然有效，最好了解这一点。

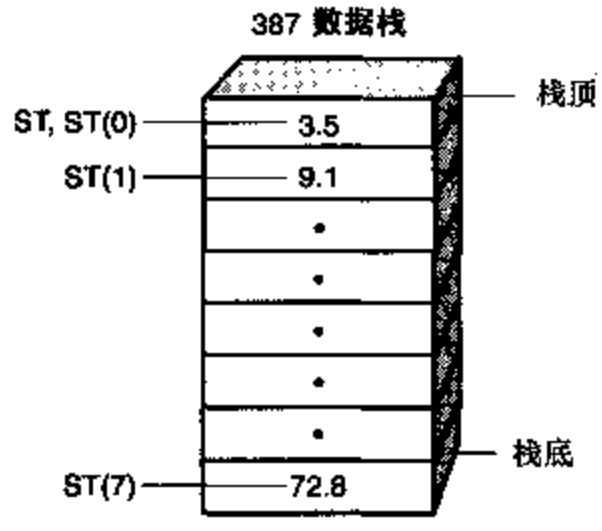


图 5.26　FPU 堆栈的排序和访问

现在读者对 FPU 的功能和工作原理有一定的了解，下面介绍 FPU 的指令集及如何使用它。

注意：因为不时地有新型号的奔腾处理器出现，所以 FPU 的指令列表也在不断增加。表 5.1 中列出大多数最常用的指令，但是您将要使用指令也不会超过其中的 10%。

5.8.3 FPU 指令集

FPU 是一个完整的处理器，有一个很大的指令集。这些指令分为以下几类：

- 数据转移指令——在 CPU 和 FPU 之间移动数据。
- 算术指令——执行标准的算术运算。
- 超越指令——执行非线性运算，如正弦、余弦等。
- 常量指令——加载常用的常量，如 1、pi 等。
- 比较指令——对浮点数进行比较。
- 控制指令——用于控制和设置 FPU 的状态。

表 5.1 列出了所有的 FPU 指令，并对每个指令的功能做了简要的描述。

表 5.1　　FPU 协处理器指令集

指令名称	描述
数据转移指令	
FBLD	加载一个 BCD 数
FBSTP	存储和弹出一个 BCD 数
FILD	加载一个整数
FIST	存储一个整数
FISTP	存储一个整数并对堆栈执行弹出操作
FLD	加载一个实数
FSTP	存储一个实数并对堆栈执行弹出操作
FXCH	交换两个栈元素
算术运算指令	
FABS	计算绝对值
FADD	实数相加
FIADD	整数相加
FADDP	实数相加并对堆栈执行弹出操作
FCHS	改变数的符号
FDIV	实数相除
FIDIV	整数相除
FDIVP	实数相除并对堆栈执行弹出操作
FDIVR	实数相除，但是交换除数和被除数
FIDIVR	整数相除，但是交换除数和被除数
FDIVRP	实数相除并对堆栈执行弹出操作，但是交换除数和被除数
FMUL	实数相乘
FIMUL	整数相乘
FMULP	实数相乘并对堆栈执行弹出操作
FPREM	计算部分余数
FPREM1	使用 IEEE 格式计算部分余数
FRNDINT	将操作数取整
FSCALE	缩放 2 的幂
FSUB	实数相减
FISUB	整数相减

续表

指令名称	描述
算术运算指令	
FSUBP	实数相减并对堆栈执行弹出操作
FSUBR	实数相减，但是交换减数与被减数
FISUBR	整数相减，但是交换减数与被减数
FSUBRP	实数相减并对堆栈执行弹出操作，但是交换减数与被减数
FSQRT	计算平方根
FXTRACT	从实数中提取指数和符号
超越数指令（角度必须以弧度为单位）	
F2XM1	计算值(2*x－1)
FCOS	计算余弦
FPATAN	计算部分反正切
FPTAN	计算部分正切
FSIN	计算正弦
FSINCOS	计算正弦和余弦
FYL2X	计算表达式 $y*\log_2 x$
FYL2XP1	计算表达式 $y*\log_2(x+1)$
常量	
FLD1	加载值 1.0
FLDL2E	加载 $\log_2 e$
FLDL2T	加载 $\log_2 10$
FLDLG2	加载 $\log_{10} 2$
FLDPI	加载 pi
FLDZ	加载零
比较运算符	
FCOM	比较实数
FCOMP	比较实数并堆栈执行弹出操作
FCOMPP	比较实数并对堆栈执行弹出操作两次
FICOM	比较整数
FICOMP	比较整数并对堆栈执行弹出操作
FIST	比较栈顶和零
FUCOM	执行无序比较（unordered compare）
FUCOMP	执行无序比较并对堆栈执行弹出操作
FUCOMPP	执行无序比较并对堆栈执行弹出操作两次
FXAM	检查 st(0)并将结果存储到条件寄存器中
控制指令	
FCLEX	清除所有未屏蔽浮点异常
FNCLEX	清除所有异常
FDECSTP	堆栈指针递减
FFREE	清除一个栈元素，使其看起来好像被弹出
FINCSTP	堆栈指针递增

续表

指令名称	描述
控制指令	
FINIT	初始化 FPU 并检查异常
FNINIT	初始化 FPU 且不检查异常
FLDCW	加载控制字
FLDENV	加载 FPU 环境
FNOP	相当于 NOP
FRSTOR	用指定内存区域恢复 FPU 的状态
FSAVE	将 FPU 的状态保存到一个内存区域中，且检查异常
FNSAVE	将 FPU 的状态保存到一个内存区域中，且不检查异常
FSTCW	保存控制字并检查异常
FNSTCW	保存控制字且不检查异常
FSTENV	保存环境并检查异常
FNSTENV	保存环境且不检查异常
FSTSW	保存状态字并检查异常
FNSTSW	保存状态字且不检查异常
FSTSW AX	保存状态字到 AX 中并检查异常
FNSTSW AX	保存状态字到 AX 中且不检查异常
WAIT	挂起 CPU，直到 FPU 完成其运算

指令有很多，无法给出每条指令的使用范例，表 5.2 给出了大多数指令的通用格式，根据这个表格，读者将能够判断出每个的指令需要什么类型的操作数。

表 5.2　　FPU 的操作数格式

指令格式类型	句法和操作数	隐含	指令格式类型	句法和操作数	隐含
经典	Finstruction	ST, ST(1)	寄存器	Finstruction ST(*n*), ST Finstruction ST, ST(*n*)	无 无
内存	Finstruction Memory_operand	ST	寄存器弹出	Finstruction ST(*n*), ST	无

注意：所有 FPU 指令都以 F 开头，且寄存器弹出指令都以字母 P 结尾。下面讨论各种指令格式，让读者知道如何在程序中使用它们。

5.8.4　经典指令格式

经典指令格式将 FPU 的堆栈视为经典堆栈。这意味着所有运算都针对栈顶 ST(0)和第二个元素 ST(1)。例如，如果使用如下指令请求执行加法运算：

```
FADD
```

将执行的运算为 ST(0) = ST(0) + ST(1)，即栈顶的值为最上面两个元素的和，如图 5.27 所示。一般来说，如果经典指令需要两个操作数，将使用 ST(0)和 ST(1)；如果只需要一个操

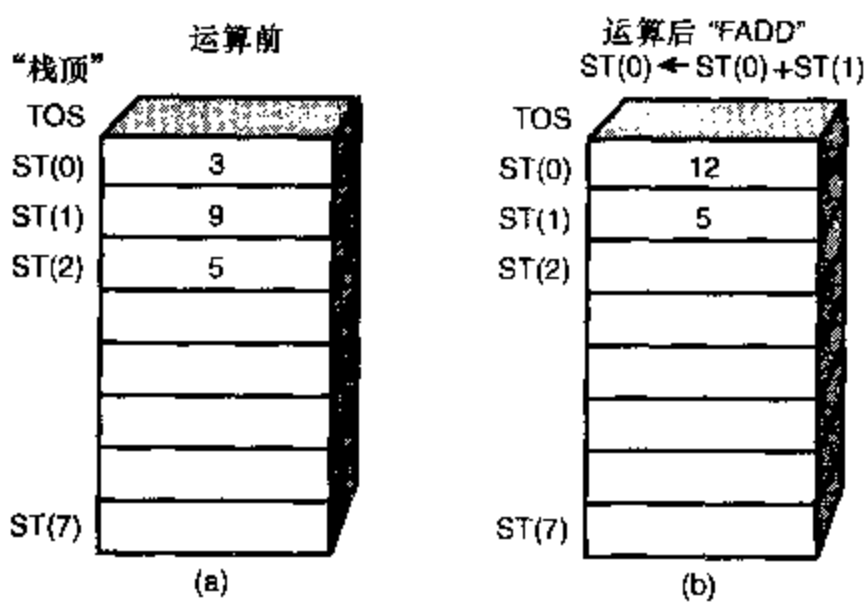


图 5.27　经典堆栈指令范例

作数，将使用 ST(0)。如果指令使用两个操作数，通常将栈顶的源操作数弹出，并将结果存储到栈顶中。

5.8.5　内存指令格式

内存指令格式与经典指令格式类似，也将 FPU 堆栈当成经典堆栈。然而，内存指令格式允许将内存项压入到堆栈中或将栈顶元素弹出到内存中。例如，可以使用如下指令将一个内存操作数加载到堆栈中：

```
FLD memory
```

要将栈顶元素弹出并存储到内存中，可以这样做：

```
FSTP memory
```

这两条指令的结果如图 5.28 所示。

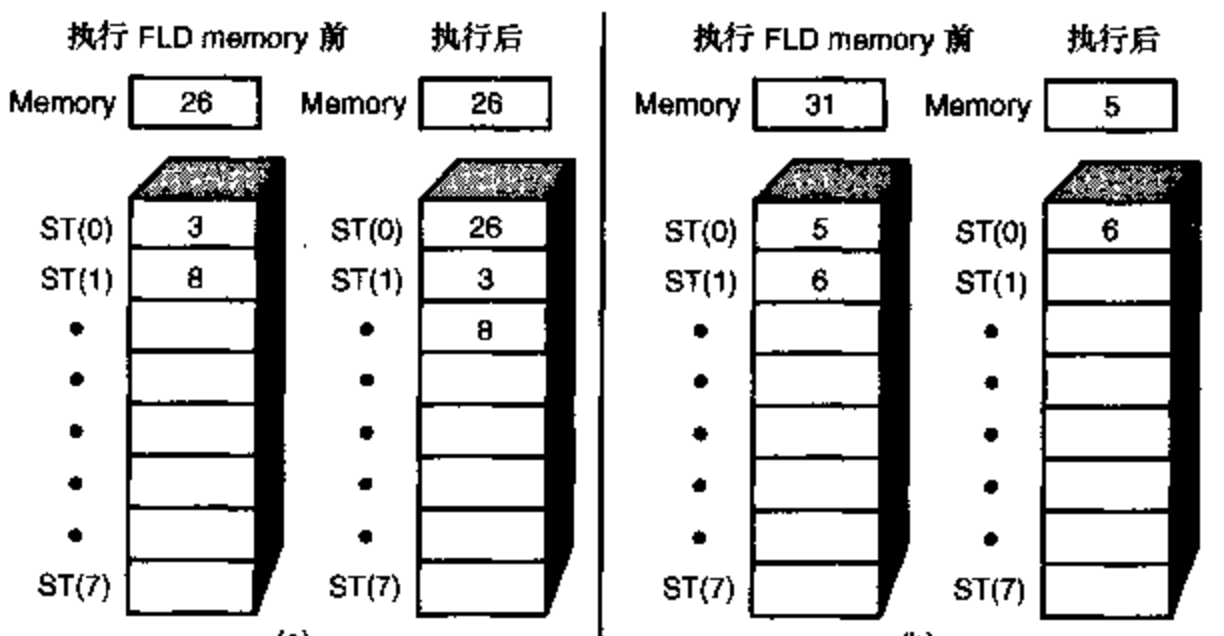


图 5.28　内存指令格式范例

5.8.6　寄存器指令格式

寄存器指令格式让您能够显式地引用栈元素（寄存器）。例如，要将 ST(0)与 ST(4)相加，可以这样做：

```
FADD ST(0), ST(4)
```

这样，ST(0)＋ST(4)的和将存储在栈顶中，如图 5.29 所示。另外，使用需要两个操作数的寄存器指令格式时，其中一个操作数必须为栈顶 ST(0)（其缩写为 ST）。

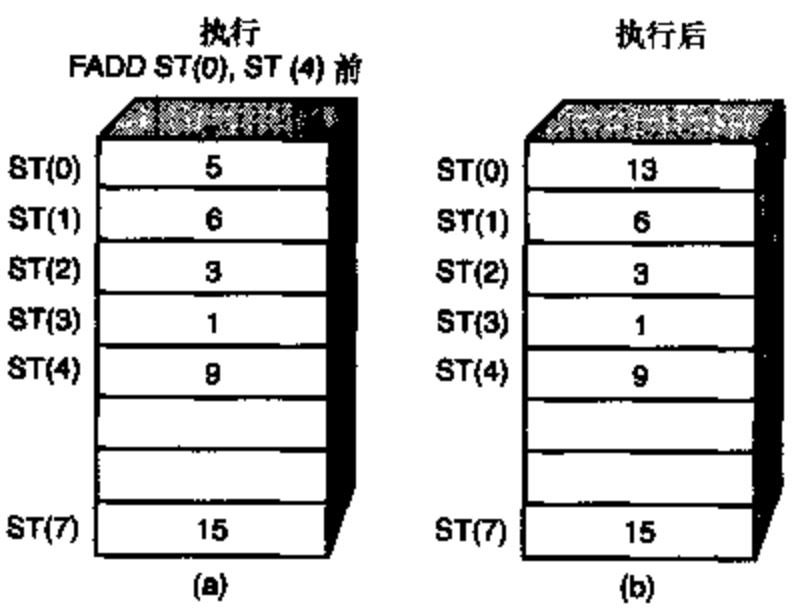


图 5.29　寄存器指令格式范例

5.8.7　寄存器弹出指令格式

寄存器弹出指令格式与寄存器指令格式相同，但将结果存储到栈顶后，对堆栈执行弹出操作一次。

5.8.8　FPU 范例

这里不打算介绍 FPU 提供的全部指令，而只讨论一些最具代表性的指令。它们是：

- FLD——全部形式。
- FST——全部形式。
- FADD——全部形式。

- FSUB——全部形式。
- FMUL——全部形式。
- FDIV——全部形式。

介绍每条指令时，包括以下内容：

1. 描述。
2. 指令句法和操作数（如果有的话）。

5.8.9 FLD 范例

各种形式的 FLD 指令用于加载实数（FLD）、整数（FILD）和常量（如 pi，FLDPI）。FLD 指令有下列格式：

- 加载一个实数：FLD op1，其中 op1 可以是 32 位短实数、64 位长实数、80 位临时实数或 ST(n)。
- 加载一个整数：FILD op1，其中 op1 可以是 16 位整数、32 位短整数、64 位长整数等。
- 加载一个常量：FLDcon，其中 con 是一个后缀，指定了要加载的常量，详情请参阅表 5.1 中关于常量的部分。

下面介绍一些范例。要从内存中加载一个 32 位实数，可以这样做：

```
FLD memory32
```

执行该指令后，栈顶 ST(0)将包含变量 memory32 的值，如图 5.30a 所示。

要将一个标准二进制 16 位整数加载到 FPU 中，可以这样做：

```
FILD memory16
```

执行该指令后，栈顶将包含用 IEEE 格式表示的这个 16 位整数，如图 5.30b 所示。

注意：浮点数处理器在处理整数时，将把它转换为实数。

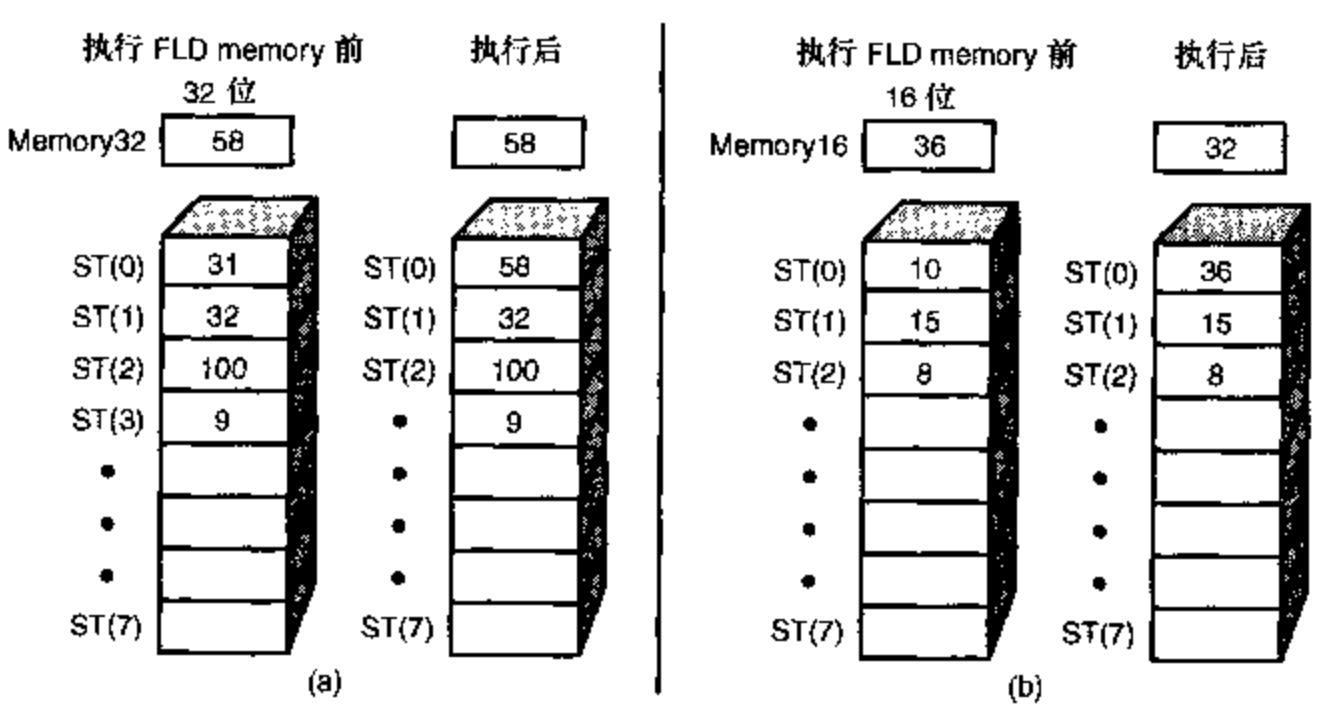


图 5.30 加载指令范例

5.8.10 FST 范例

FST 指令用于存储实数（FST）和整数（FIST）。在这两条指令后可以加上字母 P，指示 FPU 在执行存储后对堆栈执行弹出操作。实质上，在指令后加上字母 P 后，栈顶元素 ST(0)将被存储到目标操作数中，且相应地调整堆栈指针。FST 指令的格式如下：

- 存储一个实数：FST op1，其中 op1 可以是 32 位短实数、64 位长实数或 ST(n)。
- 存储一个实数并对堆栈执行弹出操作：FSTP op1，其中 op1 可以是 32 位短实数、64 位长实数或 ST(n)。

- 对栈顶元素取整并存储结果：FIST op1，其中 op1 可以是 16 位整数、32 位短整数或 64 位长整数。
- 对栈顶取整，然后弹出并存储结果：FISTP op1，其中 op1 可以是 16 位整数、32 位短整数或 64 位长整数。

另外，栈顶元素总是隐含为源操作数，因此 op1 是目标操作数。下面介绍一些范例。要将栈顶元素存储到一个标准浮点数变量中，可以这样做：

```
FST memory32
```

结果如图 5.31（a）所示。注意，栈顶元素未被修改。如果执行如下指令：

```
FSTP memory32
```

栈顶元素将被弹出到 memory32 中。下面来看一些整数范例。要将栈顶元素存储到一个 32 位整数变量中，可以这样做：

```
FIST memory32
```

栈顶元素的值将被取整，转换为一个整数，然后存储到 memory32 中，如图 5.31（b）所示。FISTP 执行相同的操作，但还对堆栈执行弹出操作，如图 5.31（c）所示。

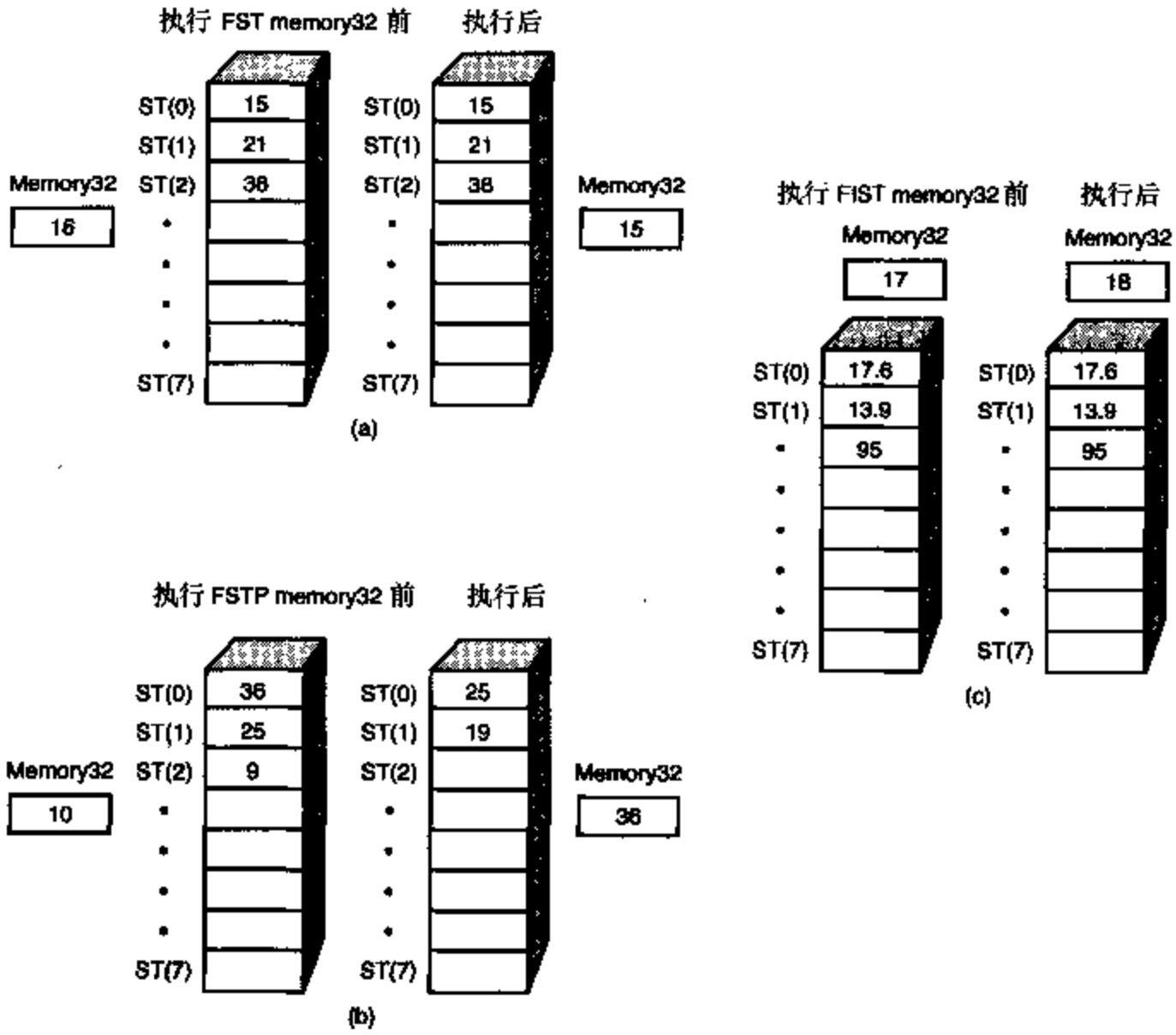


图 5.31　存储指令范例

5.8.11　FADD 范例

FADD 指令将两个实数相加（FADD）或将两个整数相加（FIADD）。另外，在实数版本指令后加字母 P 时，将在执行加法后对堆栈执行弹出操作。然而，对于整数版本，并不支持这项功能。FADD 指令有如下格式：

- 操作数默认为 ST(0) 和 ST(1)，执行加法后，对堆栈执行一次弹出操作；

```
FADD
```

● 栈顶元素默认为目标操作数：

```
FADD    op1
        memory64
        ST(n)
```

● 一个操作数为栈顶元素，但不一定是目标操作数：

```
FADD    op1, op2
        ST(n), ST
        ST, ST(n)
```

● 将两个操作数相加，然后对堆栈执行一次弹出操作：

```
FADDP   op1, op2
        ST(n), ST
        ST, ST(n)
```

● op1 引用的整数被转换为实数，然后与栈顶元素相加。

```
FIADD   op1
        memory16
        memory32
```

这些看起来非常容易。下面介绍一些范例。首先，将堆栈最上面两个元素相加，下面两个指令都能完成这项任务：

```
FADD ST(1)
FADD ST,ST(1)
```

图 5.32a 是执行上述指令后的结果。如图中所示，栈顶元素的值将设置为相加得到的和。

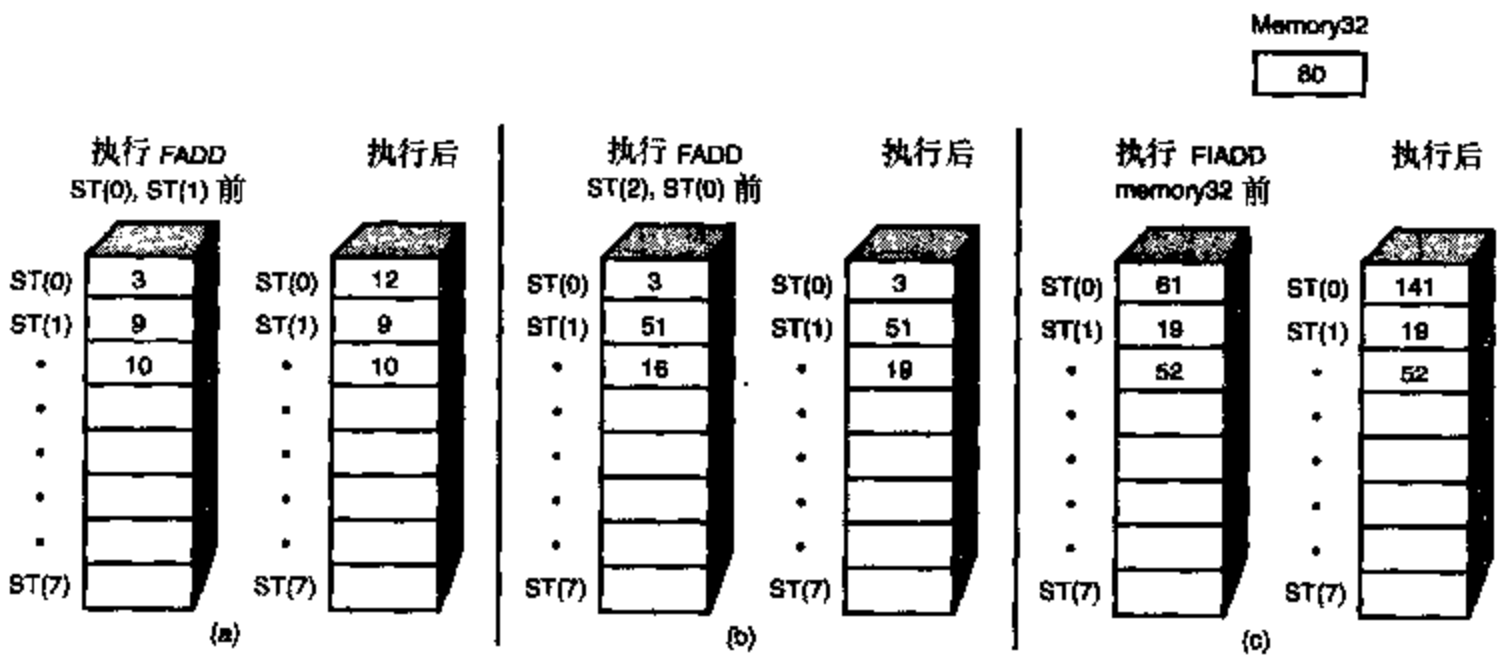


图 5.32 加法指令范例

然而，如果使用下面的指令：

```
FADD ST(2), ST
```

ST(0)与 ST(2)的和将存储到 ST(2)中，因为 ST(2)为目标操作数，而 ST(0)的值将保持不变，如图 5.32b 所示。接下来介绍一个整数形式的例子。要将内存中的一个 32 位整数与栈顶元素 ST(0)相加，可使

用如下指令：

```
FIADD memory32
```

memory32 的值将被转换为实数，然后与 ST(0) 相加。结果被存储到 ST(0) 中，如图 5.32c 所示。然而，如果要在执行加法后弹出栈顶元素，必须使用实数加法，因为不支持在执行整数加法后弹出栈顶元素。然而，可以使用如下指令手工地弹出栈顶元素：

```
FINCSTP
```

下面介绍一个小程序，它将两个实数相加，并将结果存储到一个 32 位整数变量中。下面是其源代码：

```
float fvalue_1 = 100.2;
float fvalue_2 = 50.5;
int ivalue_1 = 0;
_asm {
   FLD fvalue_1  // 加载第一个 4 字节实数
   FADD fvalue_2  // 将第二个 4 字节实数与栈顶元素相加，并将结果存储到栈顶元素中
   FISTP ivalue_1 // 将栈顶元素存储到整型变量 ivalue_1 中
            // byte integer
   } // end asm
```

图 5.33 说明了该程序的执行过程。

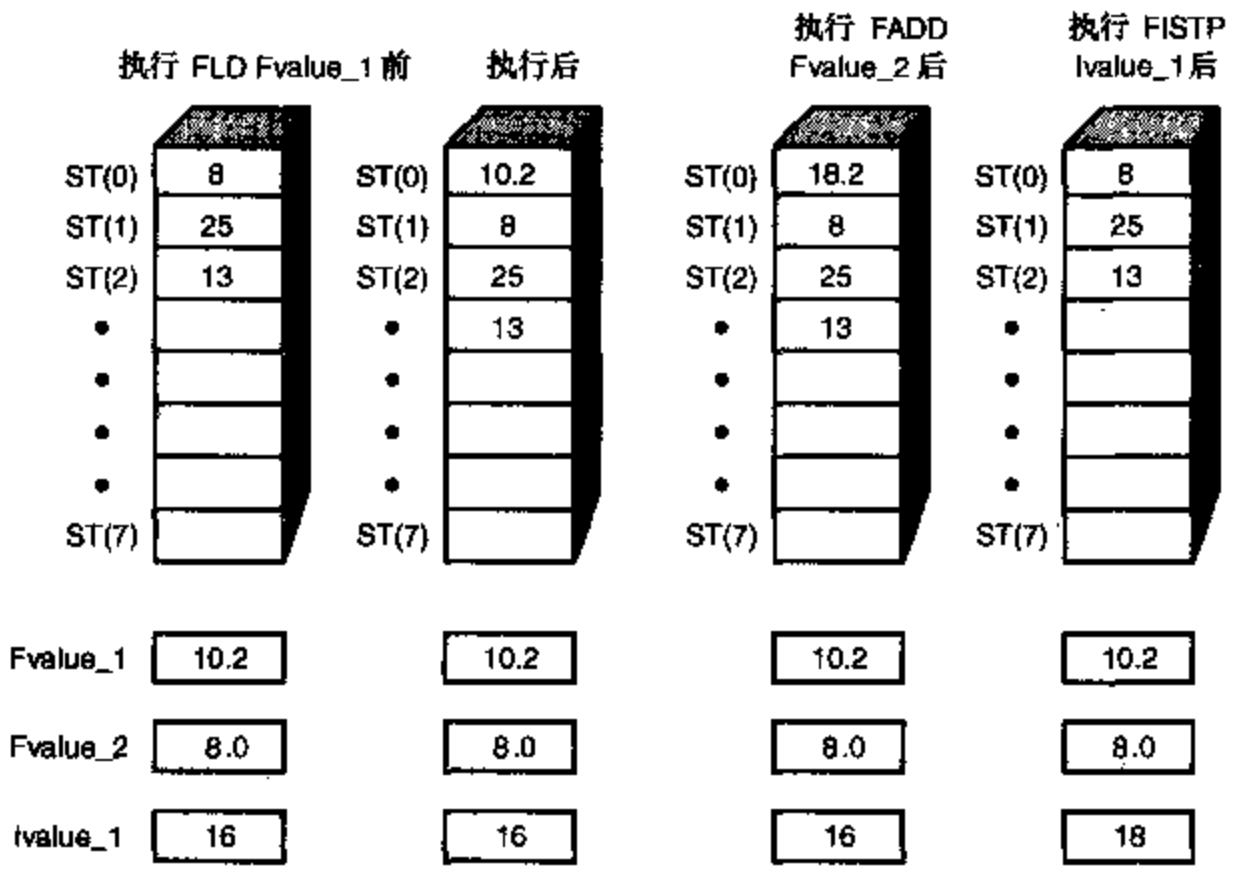


图 5.33　一个浮点数程序的执行过程和堆栈的变化

5.8.12　FSUB 范例

- 隐含地执行(ST(1)－ST(0))，且执行减法后将栈顶元素弹出。

```
FSUB
```

- 栈顶元素隐含为目标操作数，且提供另一个操作数 op1，执行的运算为(ST(0)－op1)：

```
FSUB    op1
        memory32
        memory64
        ST(n)
```

● 其中一个操作数为栈顶元素 ST，但它不一定是目标操作数，执行的运算为(op1 - op2)。

```
FSUB    op1, op2
        ST, ST(n)
        ST(n), ST
```

● 计算表达式(op1 - op2)，并弹出栈顶元素。

```
FSUBP   op1, op2
        ST, ST(n)
        ST(n), ST
```

● 将 op1 引用的操作数转换为实数，并将栈顶元素同其相减。当然，结果将存储在 ST(0)中。

```
FISUB   op1
        memory16
        memory32
```

减法与加法的工作方式相同，但源操作数和目标操作数更重要，因为(a - b)与(b - a)的结果不同。也就是说，执行减法时需要特别注意操作数的顺序。例如，下面的指令：

```
FSUB memory32
```

从栈顶元素 ST(0)中减去 memory32。即 ST(0) = ST(0) - memory32，如图 5.34a 所示。下面再来看另一个范例，假设要从栈顶元素 ST(0)中减去一个整数（可能为了是计算小数部分），可以使用下面的指令：

```
FISUB memory16
```

栈顶元素的值将为 ST(0) -(real) (memory16)，如图 5.34b 所示。同样，在计算之前，所有整数被转换为实数。

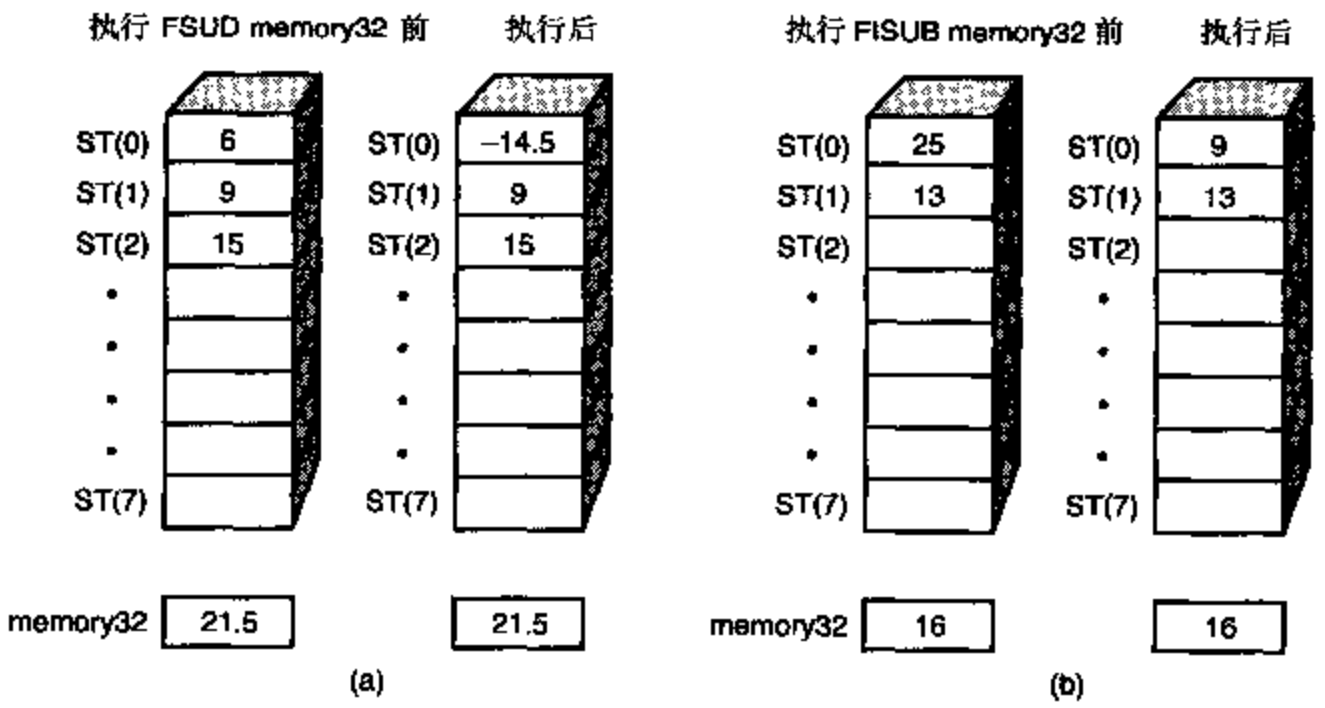


图 5.34 减法指令范例

5.8.13 FMUL 范例

FMUL 指令与 FADD 指令非常类似，可用于将两个实数相乘（FMUL），将一个整数与一个实数相乘（FIMUL），也可以在指令后面添加后缀 P，表示执行乘法后弹出栈顶元素。FMUL 指令有如下形式：

● 隐含执行运算 ST(0) * ST(1)，且执行乘法后弹出栈顶元素。

FMUL

- 栈顶元素隐含作为目标操作数，同时指定另一个操作数。

```
FMUL    op1
        memory32
        memory64
        ST(n)
```

- 栈顶元素为操作数之一，但它不一定是目标操作数。

```
FMUL    op1, op2
        ST, ST(n)
        ST(n), ST
```

- 将两个操作数相乘，并弹出栈顶元素。

```
FMULP   op1, op2
        ST, ST(n)
        ST(n), ST
```

- 将 op1 引用的整数被转换为实数，然后将其与栈顶元素相乘。

```
FIMUL   op1
        memory16
        memory32
```

假设要计算 5 的阶乘 5!，即 5*4*3*2*1，完成该任务的小程序如下：

```
_asm {
   FLD f_1 // 将 1.0 加载到堆栈中
   FLD f_2 // 将 1.0 加载到堆栈中
   FLD f_3 // 将 1.0 加载到堆栈中
   FLD f_4 // 将 1.0 加载到堆栈中
   FLD f_5 // 将 1.0 加载到堆栈中
   FMUL    // 5*4
   FMUL    // 5*4*3
   FMUL    // 5*4*3*2
   FMUL    // 5*4*3*2*1

   // 栈顶元素 ST(0)的值为 5*4*3*2*1
   } //end asm
```

图 5.35 说明该程序运行过程中的阶乘处理和堆栈。

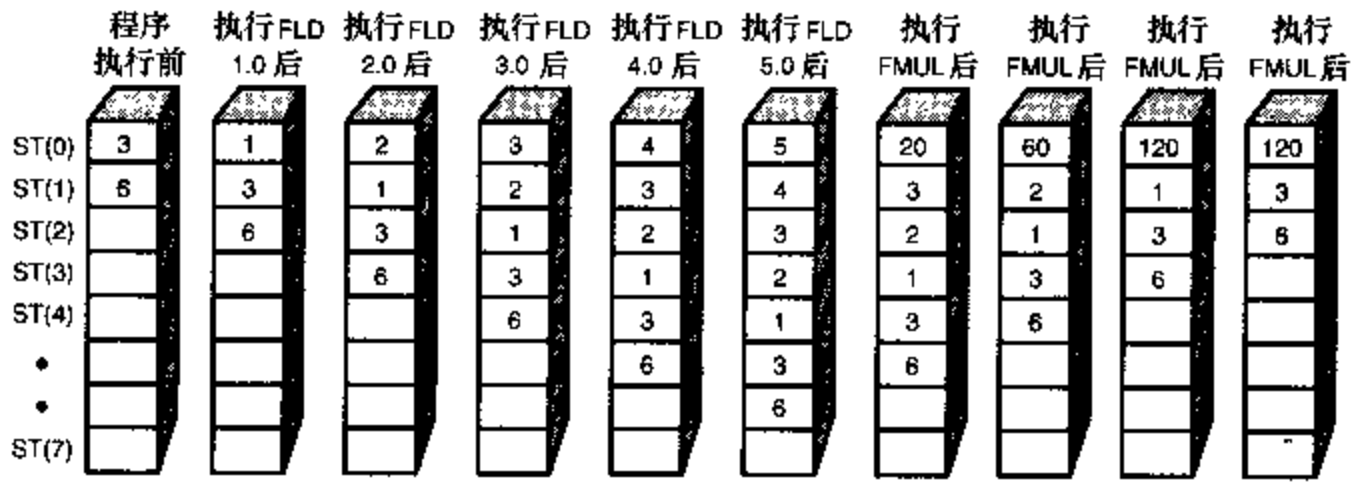


图 5.35　阶乘计算过程中的堆栈

5.8.14 FDIV 范例

我们要介绍的最后一种指令是除法指令：FDIV。它与减法指令类似，操作数的顺序将影响计算结果：即 a/b 与 b/a 是不相同的。如您期望的那样，它有全部的形式：实数、整数、和栈顶元素弹出。下面是除法指令的各种格式：

- 隐含执行运算 ST（1）/ST（0），且在执行除法后弹出栈顶元素。

```
FDIV
```

- 栈顶元素隐含为目标操作数，且指定另外一个操作数。

```
FDIV    op1
        memory32
        memory64
        ST(n)
```

- 栈顶元素为操作数之一，但不一定是目标操作数。

```
FDIV    op1, op2
        ST, ST(n)
        ST(n), ST
```

- 执行运算 op1/op2，且弹出栈顶元素。

```
FDIVP   op1, op2
        ST, ST(n)
        ST(n), ST
```

- 将 op1 引用的整数转换为实数，然后用它除栈顶元素。

```
FIDIV   op1
        memory16
        memory32
```

作为一个范例，使用如下公式：

```
velocity = distance/time
```

来计算使用火箭引擎到达月球所需要的时间：

```
// 到月亮的距离为 250000 英里
float f_dist = 250000.0;
// 太空飞船的速度为 35000 英里/小时
float f_vel = 35000;

// 时间
float f_time;

_asm {
   FLD f_dist   // 加载距离
   FDIV f_vel    // 除以速度
   FSTP f_time   // 弹出栈顶元素中的结果，并将其存储到 f_time 中
   } // end asm

printf("\ntime =%f",f_time);
```

5.9　数学引擎使用说明

在程序中使用数学引擎与使用任何其他库模块的方法相同。然而，在使用数学引擎时有点需要注意。首先，该数学引擎基于 T3DLIB1.CPP|H 中的一些代码，因此必须在使用数学库的程序中包含头文件 T3DLIB1.H 和 C++源代码文件 T3DLIB1.CPP。另外，T3DLIB1.CPP|H 以 Direct×8.0+为基础，因此必须包含 DDRAW.H 文件以获得类型信息。创建独立的控制台数学应用程序时，至少需要如下文件：

- 源文件——T3DLIB1.CPP、T3DLIB4.CPP。
- 头文件——T3DLIB1.H、T3DLIB4.H、DDRAW.H。

当然，还需要包含通常所需的头文件。不要忘记#define INITGUID；否则，链接器将无法找到与 DirectDraw 相关内容，即使应用程序没有使用任何与 DirectDraw 相关的功能，也不能通过链接。别忘了，T3DLIB1.CPP|H 需要 DirectDraw 类型，而 T3DLIB4.CPP|H 需要 T3DLIB1.CPP|H。例如，我用于开发数学引擎的测试应用程序是一个控制台应用程序，它的头文件如下：

```
#define WIN32_LEAN_AND_MEAN

#ifndef INITGUID
#define INITGUID    // 需要该语句，也可以包含文件 DXGUID.LIB
#endif

#include <windows.h>  // 包含重要的 Windows 功能
#include <windowsx.h>
#include <mmsystem.h>
#include <objbase.h>
#include <iostream.h> // 包含重要的 C/C++功能
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>
#include <direct.h>
#include <wchar.h>

#include <ddraw.h>   // T3DLIB1.H 需要它
#include "T3DLIB1.H"  // T3DLIB4 使用了其中的一些常量
#include "T3DLIB4.H"
```

当然，您的大多数应用程序都将使用整个游戏库，所以应该包含如下文件：

- 源文件：
  - T3DLIB1.CPP：图形和实用程序支持；
  - T3DLIB2.CPP：输入支持；
  - T3DLIB3.CPP：声音和音乐支持；
  - T3DLIB4.CPP：数学引擎。
- 头文件：
  - T3DLIB1.H；
  - T3DLIB2.H；
  - T3DLIB3.H；

— T3DLIB4.H。

- 库文件：
  - — DDRAW.LIB：DirectDraw；
  - — DSOUND.LIB：DirectSound；
  - — DINPUT.H：DirectInput；
  - — DINPUT8.H：DirectInput；
  - — DMKSCTRL.H：DirectMusic；
  - — DMUSICI.H：DirectMusic；
  - — DMUSICC.H：DirectMusic；
  - — DMUSICF.H：DirectMusic；
  - — WINMM.LIB：Windows 多媒体扩展（仅适用于 Visual C++用户）。

这里不介绍如何编译 DirectX 应用程序，详情请参阅附录 B。

游戏控制台

由于我们有了一个新的库模块，所以应该更新游戏控制台模块，以反映出新增的头文件。新的游戏控制台名为 T3DCONSOLE3.CPP，除新增了下面一行代码外，该文件与 T3DCONSOLE2.CPP 完全相同：

```
#include "T3DLIB4.h"
```

5.10 关于数学优化的说明

本章的源代码使用了较优化的算法；即使用泰勒级数来计算距离，使用共用系数扩展来计算矩阵的行列数等。对于它们，可以提高速度的方法有，在函数实现中删除循环，使用代数来简化计算，收集共用项等。另外，在很多算法中，我首先计算除数的倒数，然后将其与被除数相乘，从而避免使用除法。

提示：除法会占用大量的计算机资源。

然而，如前面所述，最好的优化方法是在算法级进行优化，或者根据已知的事实进行假设，从而提高速度。例如，假设要执行大量的矩阵乘法。可以很容易地分析出，与大多数变换矩阵相乘时，将执行很多无用的乘法，因为它们大多是稀疏矩阵，即矩阵中有很多零。这是一种常见的优化。然而，在有些时候，没有办法让算法的速度更快，这时使用内联汇编语言是惟一的优化方法。然而，就本书而言，为得到所需的性能，我怀疑是否真正需要使用内联汇编语言。我对 C/C++编译器的优化程序寄予了厚望！

总之，就优化而言，应首先编写清晰的代码和健壮的算法，最后才进行低级优化。

5.11 总　　结

本章介绍了很多基础知识，列出了大量的代码。通常，我并不喜欢列出这样多的代码，但这毕竟是一本关于编程的书籍，本章的所有内容都只是前一章内容的 C/C++版本。只有头文件中的宏没有逐行介绍过，但您应该能够理解它们的用途。当您编写使用该数学引擎的代码时，需要的一些函数和宏很可能我已经编写好了，因此在自己动手编写函数和宏之前，应确认它们是否已经有了。

在不远的将来，我们将面临的惟一问题是表示方法，即应使用 3D 点还是齐次 4D 点？我们将拭目以待，

但至少我们知道如何处理这两者。实际上，我们将使用 4×3 和 4×4 矩阵，所以从技术上说，我们将使用 4D 齐次点，并假设 w=1 或将 w 作为第 4 个分量。

最后，就编写本章而言，TI-89/92 计算器提供的帮助非常大，它可以帮助我检查工作。如果您计划编写一些数学函数，强烈推荐您购买一个可执行矩阵、向量和符号计算的计算器，以便可以用测试数据来检查您的算法。编写四元数代码时，它的帮助将更大，因为四元数没有任何可视化类比；即它完全是一种数学概念。

第 6 章　3D 图形学简介

本章将介绍大量有关 3D 图形学的基本知识，读者可将其视为包括细节的总结。接下来的各章将分别着重阐述 3D 图形学的各个主要专门领域。作为绪论，本章不深入探讨晦涩难懂的内容，而只是 3D 图形学的速成课程，让读者在阅读全书之前对 3D 图形学的整个处理流程有所了解。然后，将探讨众多的主题，其中一些涉及大量的细节，而有些几乎不涉及任何细节。阅读本章后，读者将对本书余下的内容有深入了解。本章包括以下内容：

- 3D 引擎原理；
- 3D 游戏引擎的结构；
- 3D 坐标系统；
- 基本的 3D 数据结构；
- 变换和基本动画；
- 观察流水线简介；
- 3D 引擎类型；
- 3D 工具；
- 加载外部数据。

6.1　3D 引擎原理

本书的主旨是讲授 3D 图形学，而不是介绍如何使用 API 和加速器。本书介绍创建 3D 图形引擎所需的基本知识，这意味着读者阅读本书后，将了解创建 3D 引擎的每个步骤——从数据表示到最后的光栅化。实际上，如果您在 NVIDIA、ATI 等公司工作，将能够创建整个引擎 API 和底层软件算法，它们是由硬件状态机实现的。另一方面，本书不打算讲授如何使用诸如 Direct3D 和 OpenGL 等 3D API，它们虽然很酷，但读者要真正地理解 3D 图形学，必须知道如何动手编写 Direct3D 和 OpenGL，这正是作者追求的目标。最起码，作者希望读者阅读本书后，能够在只知道视频缓存地址的情况下，在任何计算机上编写出基于软件的 3D 引擎。在具备这种能力的情况下，学习 3D API 将是小菜一碟。

6.2　3D 游戏引擎的结构

编写 3D 引擎涉及的工作比仅仅编写 3D 引擎多得多。作为游戏程序员，编写 3D 引擎的目的是使用它来渲染 3D 游戏，这意味着必须将游戏引擎、物理系统、AI 以及大部分游戏内容同 3D 引擎本身集成起来；否则将问题重重。因此，编写实际的 3D 引擎时，需要考虑它与游戏引擎之间的关系以及它们如何交互，否则将一团糟。图 6.1 说明了 3D 游戏的组成部分：

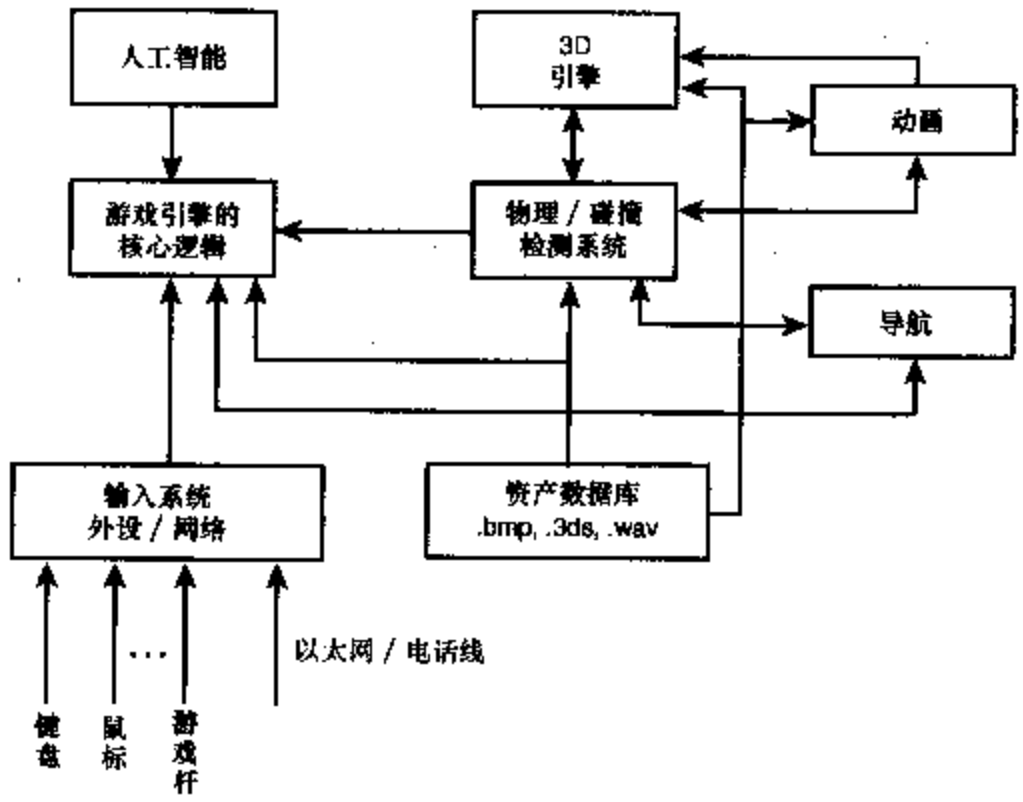


图 6.1　3D 游戏引擎的结构

- 3D 引擎；
- 游戏引擎；
- 输入系统和网络；
- 动画系统；
- 碰撞检测和导航系统；
- 物理引擎；
- 人工智能系统；
- 3D 模型和图像数据库。

正如读者看到的，需要考虑的因素很多。在简单的 2D 游戏中，碰撞检测、人工智能（AI）和数据库等问题都非常简单；但在 3D 游戏中，必须考虑这些问题。本书的重点是 3D 引擎，但显然将涉及其他问题。接下来讨论这些问题及其需要注意的事项。

6.2.1　3D 引擎

3D 引擎本身是一种软件，负责处理 3D 世界的数据结构（包括所有的光源、行动和常规状态信息）以及从玩家或相机所在的视点渲染 3D 世界。以模块化方式设计时，引擎将相对简单些，因为在这种情况下，3D 引擎和其他游戏子系统之间的交互较少。编写 3D 引擎时，可能犯的最大错误之一是，除了 3D 渲染外，还让它负责完成其他工作。例如，除处理 3D 图形外，还在 3D 引擎代码中完成物理造型（modeling）工作，这样做无疑是大错特错。另一种错误是在 3D 引擎中完成一些联网工作以简化网络，这是编写 3D 引擎的大忌。

提示：3D 引擎只负责一项工作——根据数据进行 3D 渲染，仅此而已。

6.2.2　游戏引擎

在 3D 游戏中，游戏引擎涵盖的内容很多，有时又几乎不涉及任何内容。然而，在大多数情况下，游戏引擎是一个处理所有事务的系统，它是一个控制模块，向所有的子系统发出命令。

因此，游戏引擎需要与每个游戏子系统联系。设计 3D 游戏时，一种错误的做法是首先设计 3D 引擎，然后将其某部分作为游戏控制中心，这种想法要不得。一开始，就应设计一个负责控制游戏中一切事务的模块，将其称为“游戏引擎”或其他任何名称。游戏引擎是一个容器，它容纳了其他所有的组件，熟悉整个系统的各个部分。

6.2.3 输入系统和网络

在 3D 游戏中，I/O 系统和网络系统以容易出现问题而著称，事实确是如此。问题在于，网络游戏需要通过网络传输的数据如此之多，以至于让 3D 网络游戏能够正常运行都是一个奇迹。图 6.2 说明了两种网络游戏模型。然而，对于 3D 游戏的网络部分，一种惯例是首先考虑游戏本身，并将网络部分和游戏分开。网络部分不应包含在 3D 引擎、AI 系统或物理系统中，而应将其完全独立出来。另外，考虑其他部分之前，应首先考虑 3D 游戏的网络支持。应首先考虑要使用什么类型的数据和同步技术来确保 3D 仿真的同步，然后再据此编写整个游戏；否则结果将一团糟。

提示：编写 2D 游戏时，如果在开发末期再加入网络支持，也许可以以代码难懂为代价获得 50% 逃脱失败命运的几率，但如果编写 3D 游戏时这样做，注定将以失败告终。因此，一开始就应做出网络支持方面的决策，然后在实现系统之前创建模型和仿真，并编写测试用例（test case）来测试它。

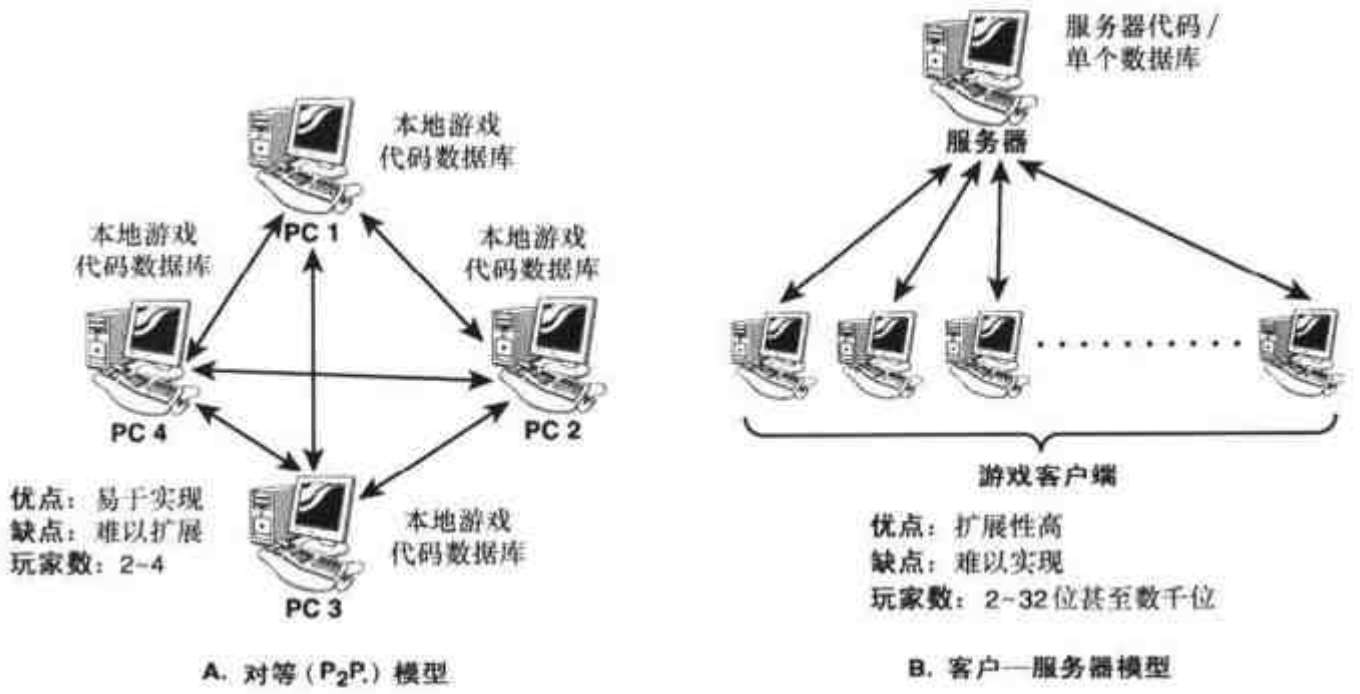


图 6.2 网络游戏模型

6.2.4 动画系统

这是与其他系统有也可能没有重大关联的重要系统之一。在 3D 游戏中，动画方式有多种：

- 简单运动；
- 复杂动画；
- 基于物理学的动画。

1. 简单运动

简单运动通常由游戏物体的基本平移和旋转组成，如图 6.3 所示。这种运动可以使用数据文件、AI、模式（pattern）、简单逻辑、状态机等来控制，没有什么神奇之处。然而，控制运动的系统可能或多或少地同其他系统相关联。例如，假设在游戏中，个体的运动是由人工智能系统控制的。这些个体在如图 6.4 所示的地图中移动。如果 AI 系统拥有描述该地图的数据，将能够执行碰撞检测，避免个体穿越墙壁。另一方面，AI 系统可能将命令传递给物理系统，而后者也能够处理碰撞检测，从而判断某种运动是否可能。正如读者看到的，至少在这个例子中，AI 系统、运动系统和物理系统位于一个联系紧密的系统中，且是可行的。

2. 复杂动画

在 3D 中，复杂动画的含义非常丰富。但通常意味着有通过关节相连的多级（articulated-hierarchical）物体，其中的链杆（link）必须相对移动，如图 6.5 所示的机器人坦克，其上半身是一个炮塔。出于简化的目的，我们首先以坦克为例，它由车体和炮塔组成。炮塔可以绕车体上面的轴（通常平行于地面）旋转，但不能脱离车体。现在的问题是，对炮塔的控制将完全基于 AI 呢还是基于数据？换句话说，我们想让炮塔

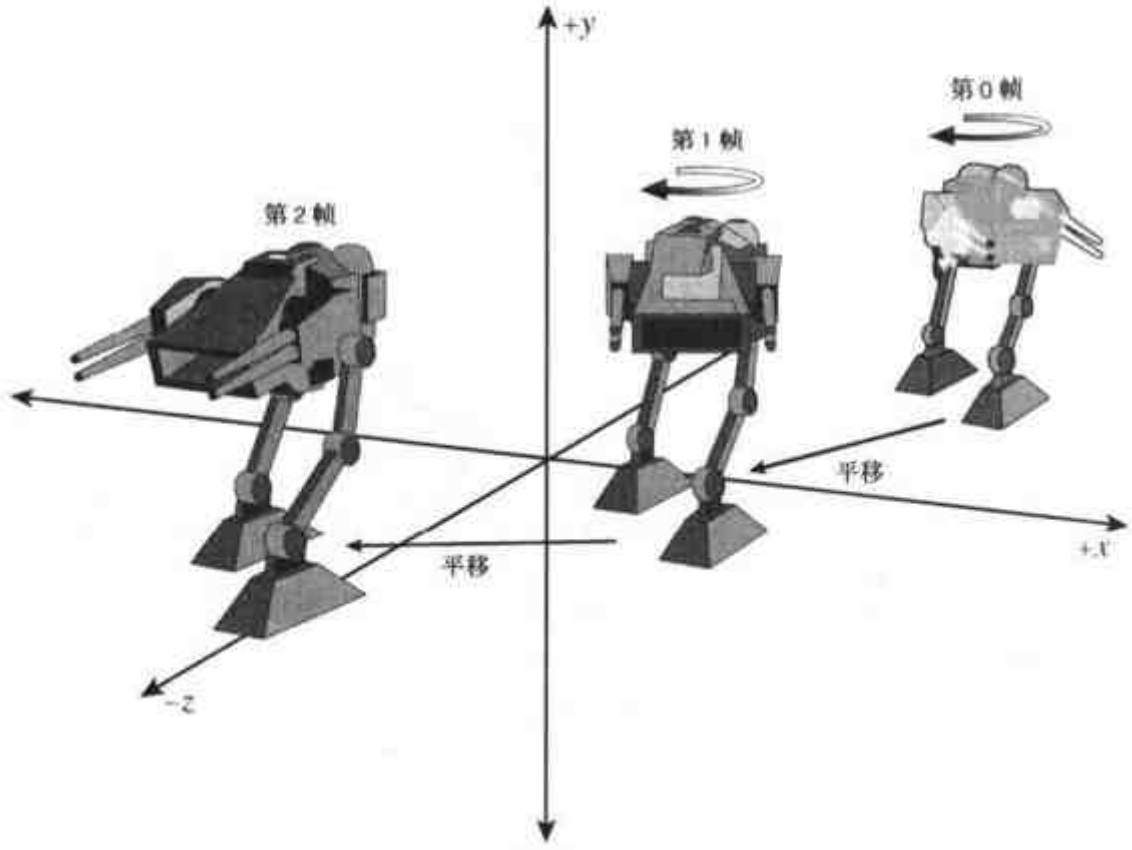


图 6.3　简单平移和旋转

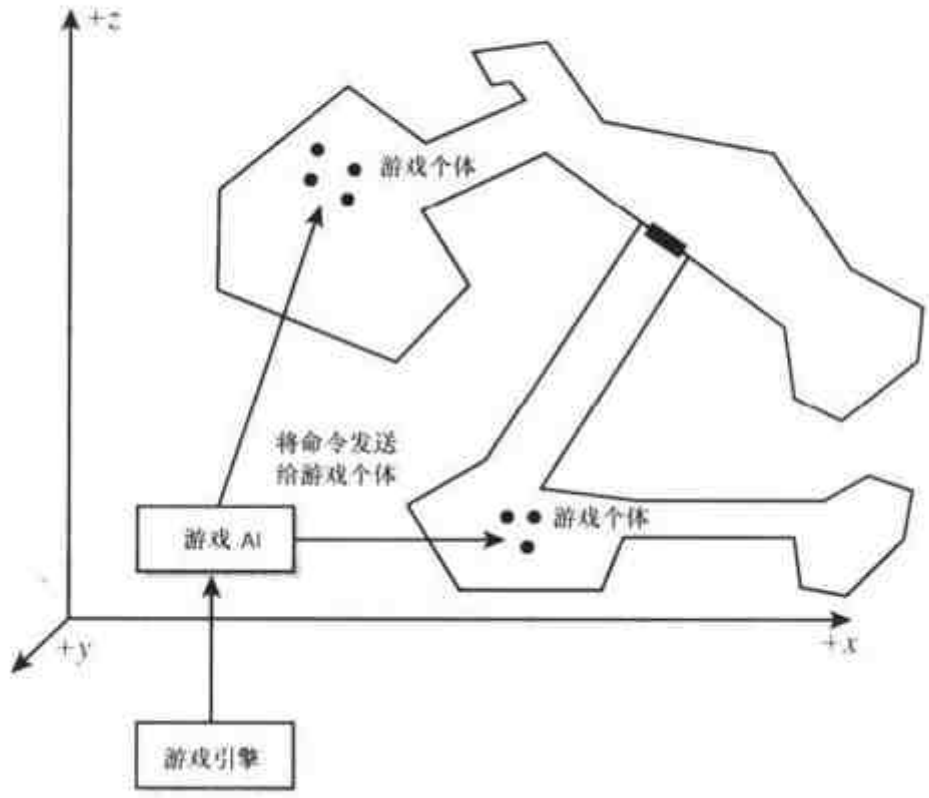


图 6.4　移动游戏个体的人工智能

进行某些有趣的运动。如果使用 AI 代码来完成这些任务，可能非常困难；但如果使用动画指令或关键帧数据，将非常容易。下面花些时间来讨论这一点。

对于由多个部件（或通过关节连接的物体）组成的物体，制作动画的方法有两种。第一种方法是，使用 3D 建模工具制作网格（mesh），然后导入这些网格，并依次加载网格 1、网格 2、网格 3 等。这种技术类似于标准的 2D 位图动画，只不过其中的数据表示的是 3D 模型。这种方法是内存密集型的，但实现起来非常容易。图 6.6 是一个这样的例子。简单地说，我们建立炮塔角度各不相同的坦克模型，然后存储网格数据输出。为制作坦克动画，我们加载这些网格，并每次渲染其中的一个。Quake 和 Quake II 使用的就是这种技术。

另一种实现 3D 物体动画的方式是使用运动数据（motion data）。一种流行的运动数据格式是 BioVision .BVH 格式，如下述代码段中所示。有关这种数据格式规范，请参阅附带光盘中本章对应目录中的文件 BIOVISION.DOC。简单地说，这些数据告诉引擎，链杆需要相对于其他链杆做什么样的移动，引擎将相应地移动 3D 物体的各个部件。这种技术灵活得多，因为创建能够理解 BioVision 数据的动画引擎后，便可以输入表示任何动作的数据，无论是漫步还是跳舞，角色（character）都将做出相应的动作。Quake Arena、Doom III、Halo 以及大部分格斗游戏（如 Tekken 系列）使用的不是这种技术。如果读者要购买运动数据，可访问网站 http://www.biovision.com。

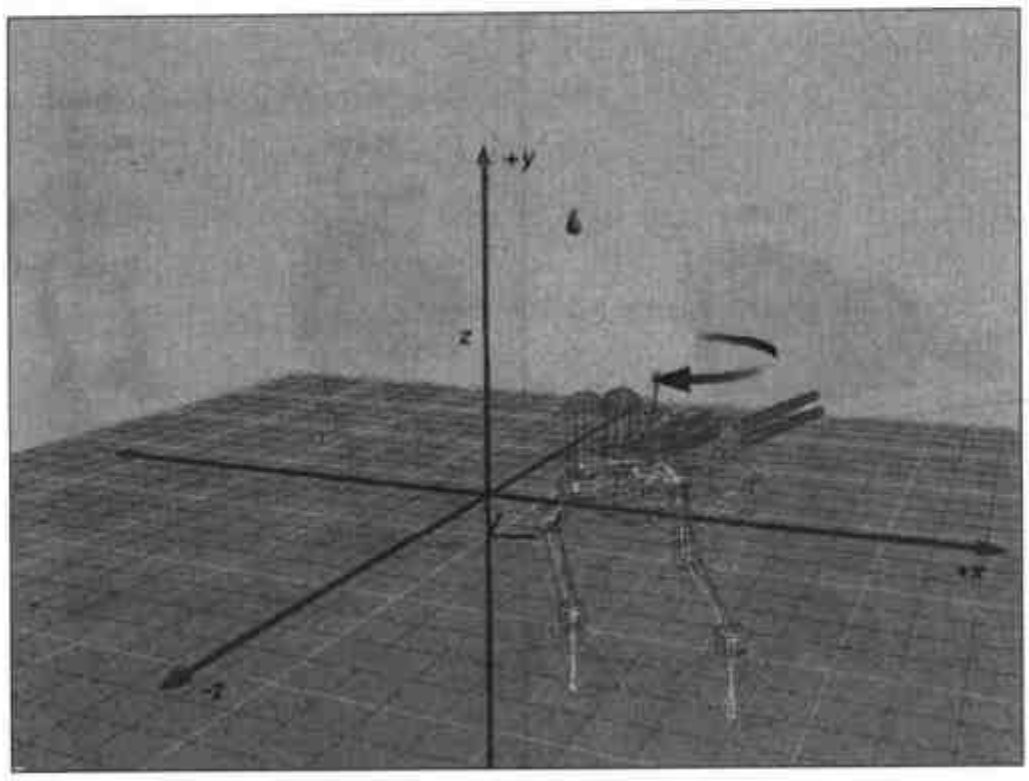


图 6.5　关节型 3D 模型动画

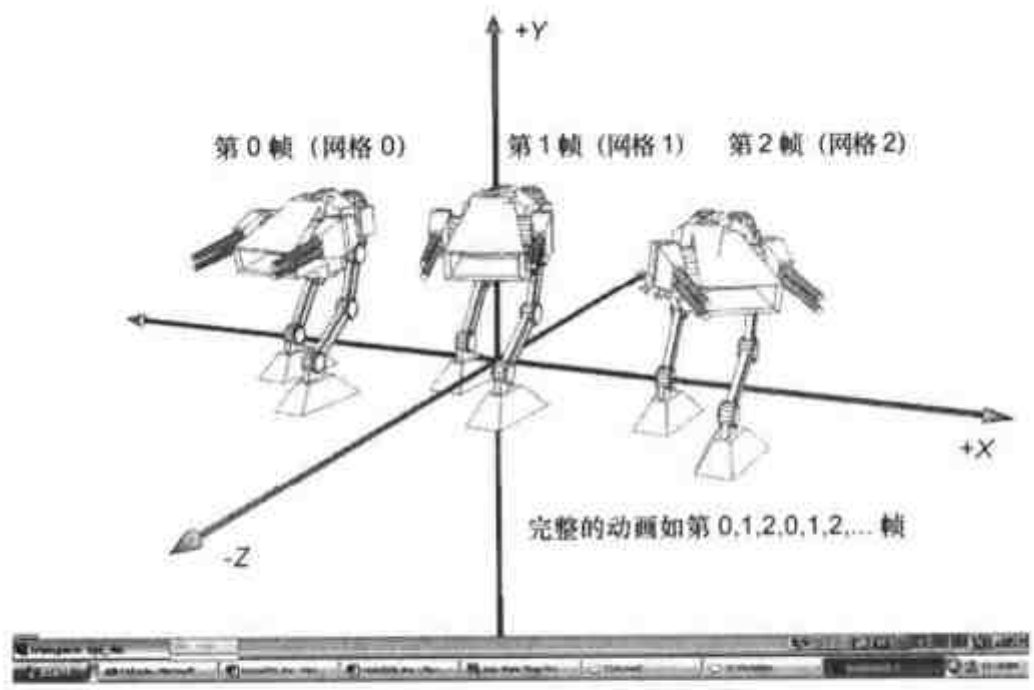


图 6.6　使用多个帧来制作动画

```
HIERARCHY
ROOT Hips
{
    OFFSET     20    0.00    0.00
    CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
    JOINT LeftHip
    {
        OFFSET     3.430000     0.000000     0.000000
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT LeftKnee
        {
            OFFSET     0.000000    -18.469999     0.000000
            CHANNELS 3 Zrotation Xrotation Yrotation
            JOINT LeftAnkle
            {
                OFFSET     0.000000    -17.950001     0.000000
                CHANNELS 3 Zrotation Xrotation Yrotation
                End Site
                {
                    OFFSET     0.000000    -3.119996     0.000000
                }
```

在 3D 动画中，需要解决的问题是，如何在反馈循环中确定限制条件，以确保动画遵循现实世界中的规律。例如，假设坦克前面是一堵墙（如图 6.7 所示），此时系统发出将炮塔逆时针旋转 90 度的命令。这

种命令可能是 AI 系统发出的，因为坦克想射击后面的目标，但 AI 系统目光没有那么远大，未考虑炮塔旋转时是否会撞向其他东西这一问题。这是 3D 游戏中常见的一种漏洞——读者肯定遇到过物体穿过其他物体的情形。之所以会出现这种问题，是由于游戏设计人员并不关心车轮是否在路面之下或拳击手的手臂是否击穿大树。然而，要避免这些情况发生，必须进行相应的检测。

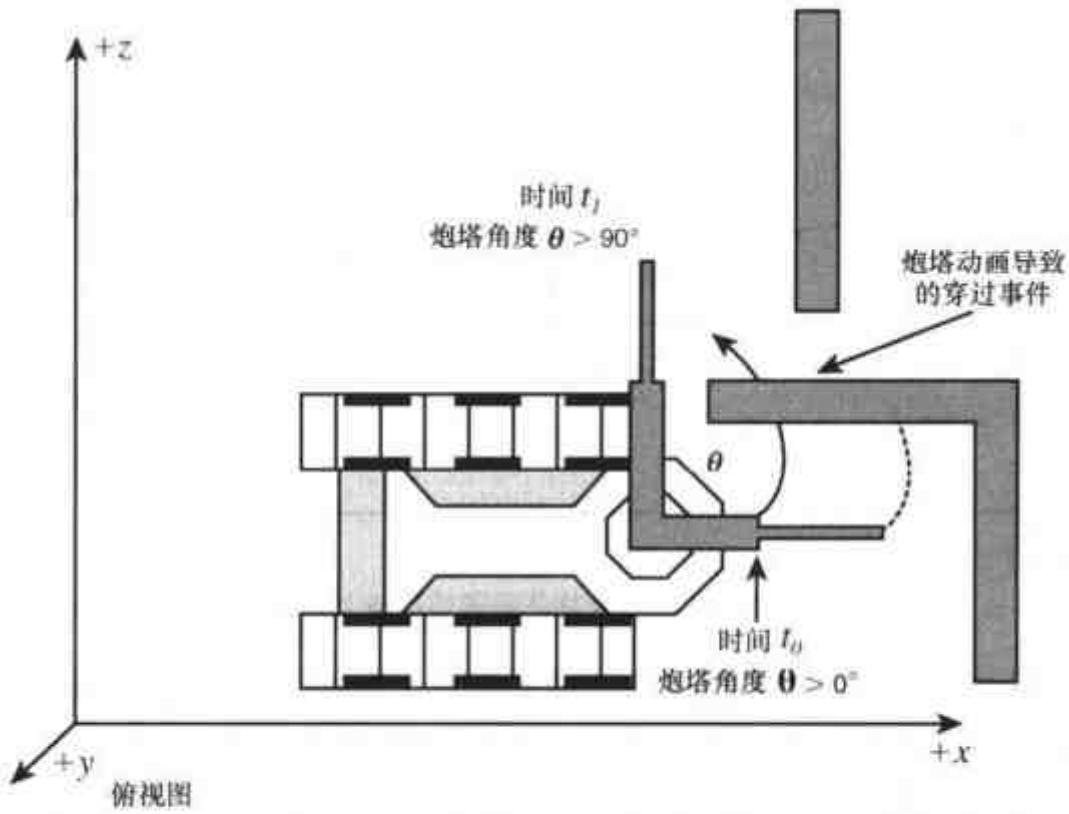


图 6.7　在不是基于物理学的动画中，出现的不符合物理学规律的情形

这种检测可能是在动画系统中进行的，也就是说，炮塔被命令进行旋转时，动画系统将检查是否会发生碰撞，据此执行或拒绝执行这种命令。这提出了另外一个问题：这种工作应由动画系统、AI 系统还是物理系统来完成？同样，这取决于很多因素。如果在游戏中，物理造型工作量非常大，动画系统应将这项工作交给物理系统去完成；否则，由动画系统来完成检测是可行的。

在 3D 游戏中，有很多复杂的关系，希望读者注意到了这一点。

3. 基于物理学的动画

这是所有动画中最复杂的。基于物理学的动画意味着物体的移动在很大程度上是由物理定律控制的。例如，在 3D 格斗游戏中，拳击动作本身可能是预先录制的运动捕捉数据（motion capture data）。然而，这些数据被输入给基于物理学的动画系统，后者控制执行动画的基于物理学的模型。它们不是盲目地旋转关节和平移物体，而是命令模型去完成动作。然后，由物理系统来判断这种动作是否可行。这是实现最逼真模拟的最佳方式。

还是以格斗游戏为例，假设拳击手 A 挥拳猛击拳击手 B。挥拳动作实际上是一组运动捕捉数据，这些数据可视为向模型的物理系统发出的命令，而不仅仅是通过模型来“播放”。因此，当拳头打到墙壁上时，物理系统将对此做出反应——通过运动控制器返回一个作用力，进而使拳头停止不动——这是符合现实的。再举一个例子，假设拳击手 B 将拳击手 A 扔了出去。扔出去的动作是运动捕捉的，但拳击手被扔出去后，处理工作将由物理系统接管，物理系统根据计算拳击手 A 的运动轨迹，并根据环境使其撞上地面或其他物体。

这里的关键之处在于，在基于物理学的动画系统中，动画系统和物理系统之间是紧密相连的。然而，它们同样与 3D 渲染毫无干系。进行渲染时，3D 引擎只需要一个多边形列表。

6.2.5　碰撞检测和导航系统

我们多次提到过物理系统，以证明 3D 游戏中的很多系统可能彼此相关联。然而在有的时候，游戏只需要一个简单的碰撞检测和导航系统，而不必创建功能完备的物理系统。我们可能将其称为物理系统，但除非它是基于力、动量或其他物理量，否则充其量只不过是一个碰撞检测系统而已。例如，创建一个 Doom

式的第一人称射击游戏（FPS）时，几乎不涉及任何物理量：角色动画是使用 AI 和预先录制（pre-canned）的网格实现的，没有滑动的东西（没有动量传递等）。惟一要考虑的是物体之间以及物体和游戏环境之间的碰撞（如图 6.8 所示）。

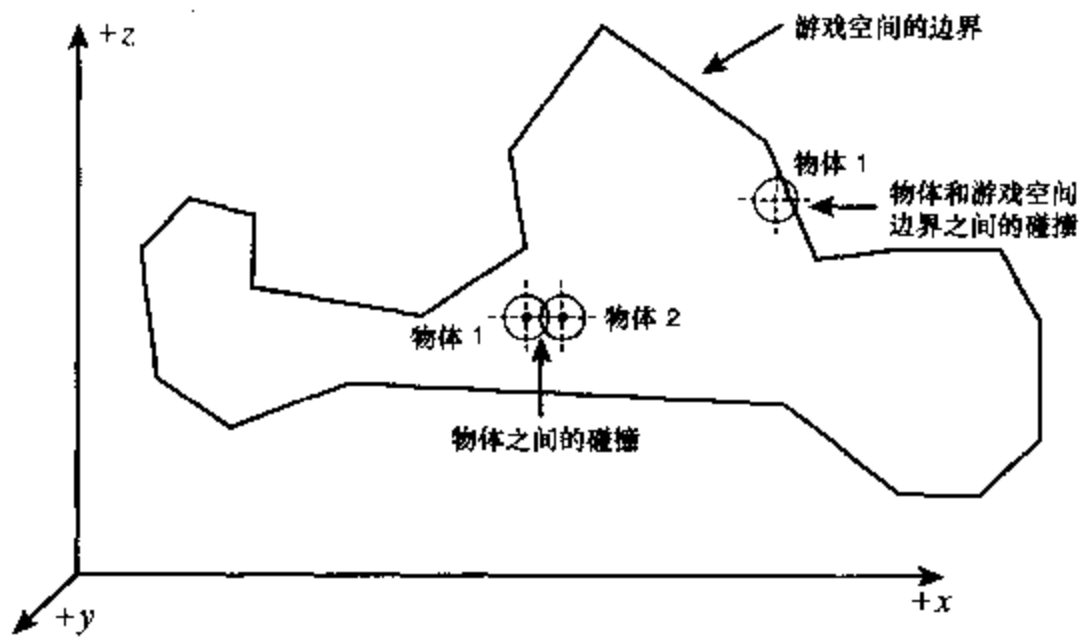


图 6.8　物体之间以及物体和游戏环境之间的碰撞

因此，只需要一个简单的碰撞检测系统。在动画系统命令物体行走、奔跑和移动时，碰撞检测系统将发生的情况反馈给动画系统（和/或 AI 系统）。然而，使用这种简单系统时，物体之间可能彼此穿过对方，怪物在地面上摔倒时可能显得不真实，等等。

6.2.6　物理引擎

游戏中的物理系统可能非常简单（如前面提到的碰撞检测系统），也可能非常复杂，这取决于实际的物理模型和闭环（closed-loop）模型。

游戏引擎中的物理系统与 3D 图形的可视化表示毫无关系。物理引擎所做的工作是，控制游戏中的物体可做出什么动作以及对动作做出响应——根据物理模型采取相应的措施。例如，非常简单的物理模型可能具有下述功能：

- 计算摩擦力和加速度；
- 弹性碰撞检测和响应；
- 线性动量传递。

在图 6.9 所示的简单 3D 赛车游戏中，赛道位于同一个平面中，其中有很多赛车，它使用的就是上述物理模型。对于每辆赛车，我们用一个点质量（或 4 个点质量，每个轮子一个）来模拟。赛车由 AI 或玩家控制，在赛道上行驶时，物理系统只根据摩擦力来确定赛车在赛道上的位置。也就是说，如果赛车的速度为 100 mph，并以某种角度转弯，物理系统将计算向心力和离心力，并据此修改赛车的加速度。另外，如果赛车相撞，将赛车视为球体，假设发生的是弹性碰撞，并据此计算动量传递，但不计算角动量（或虚构角动量）。

对于很多 2D 游戏和 3D 游戏而言，上述级别的物理模型足够了；但如果要求逼真，则需要改进。这意味着需要根据物体的实际几何形状对其进行建模，同时需要使用完整的物理模型：由物理系统控制物体；而不是由 AI 系统控制物体，并从简化的碰撞检测和摩擦力计算系统哪里接受反馈信息。在完整的物理模型中，每个物体都不断被模拟，作用到物体上的所有力都被用于标准的物理方程中，这些方程计算加速度、速度和位移。通过积分，这些方程被用于计算物体的位置：根据作用于物体上的力，将物体的位置表示为一个时间函数。这正是动力学的作用所在。

在本书后面介绍建模时将相当详细地讨论这一点。然而别忘了，在采用这种模拟等级的 3D 游戏中，动画系统和 AI 系统必须与物理系统紧密联系，这样其他系统才能对物理模型实现的人工逼真（artificial reality）做出响应，而不会不断地向游戏个体发出由于物理学约束条件而不可能执行的命令。

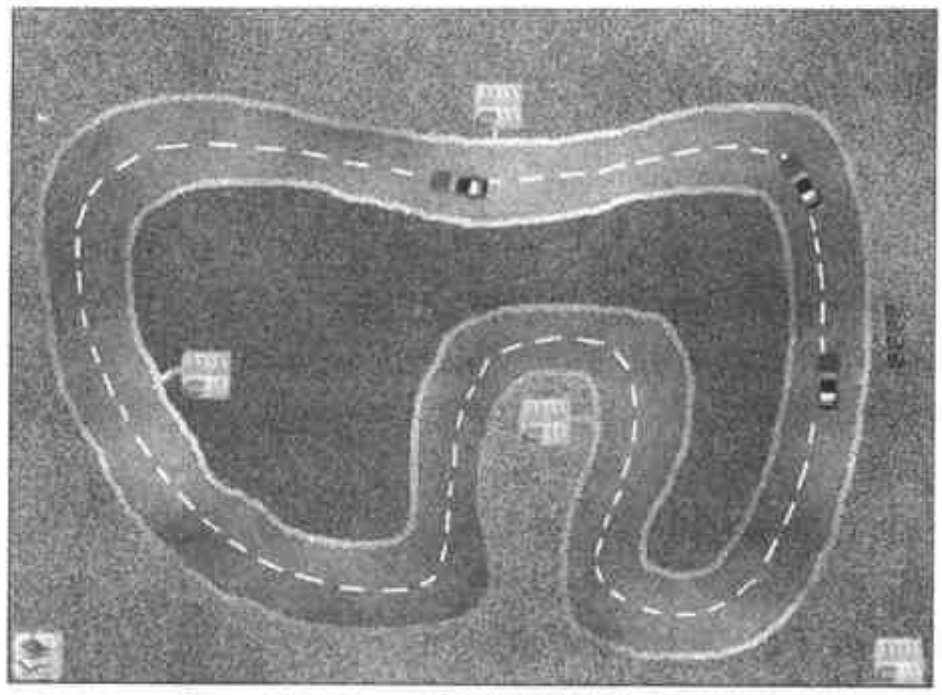


图 6.9　一个简单的赛车游戏

6.2.7　人工智能系统

3D 游戏中，AI 系统的复杂度可能比 2D 游戏高一个数量级，也可能差不多，这取决于游戏个体的建模方法。当然，在 3D 环境中，AI 存在的问题在于环境是 3D 的。诸如导航、寻径和动画等函数要复杂得多。在 2D 游戏中，移动一系列宇宙飞船也不那么困难，但在 3D 空间中移动任何东西都是困难的。3D 中使用的算法与 2D 相同：确定性、模式、规划系统、有限状态机、模糊逻辑、神经元网络等，只不过实现这些算法时需要考虑的因素多得多。

例如，图 6.10a 和 6.10b 分别是射击游戏 Valkyrie 和 Quake III 的 2D 环境和 3D 环境。该 2D 游戏的 AI 系统非常简单，因为环境很简单。其中的物体只不过是 2D 位图动画，大多数物体运动是由模式、跟踪逻辑和有限状态机控制的。3D 游戏的运行方式与此类似，但编写难度要高 10 倍。问题在于，一切都在 3D 空间内，即使是角色移动也是一个大问题。3D 角色接近玩家时，必须显得很聪明。这意味着其运动、动画和逻辑必须考虑到 3D 环境。

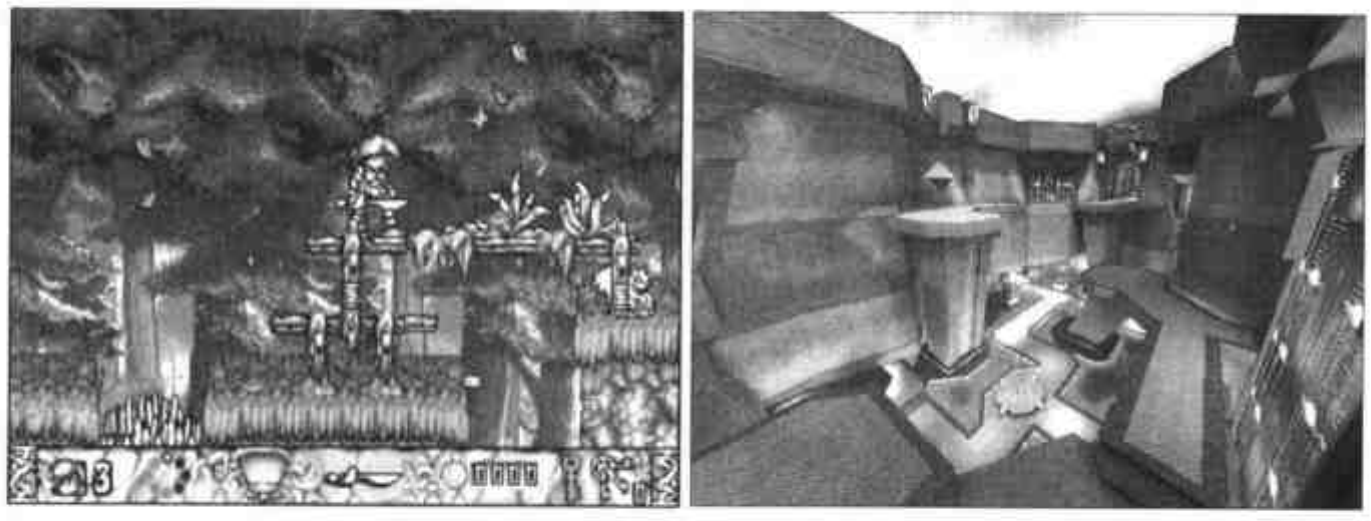


图 6.10　2D 环境和 3D 环境

这些约束条件使得编写 3D 游戏 AI 要难得多。由于 3D 世界的几何结构极其复杂，在其中将个体从一点移到另一点涉及的计算量都可能非常大。对人而言，发现一条简单路径可能很容易，但对 AI 来说，发现这样的路径一点也不容易。总之，这些都是 3D AI 编程中将出现的问题。最后，在 3D 游戏中，AI 系统依赖于动画系统和物理系统，因为 AI 需要来自游戏世界的输入，这常常是对其操作做出的响应。

6.2.8　3D 模型和图像数据库

3D 游戏的最后一个组成部分是数据，这包括很多。鉴于其 3D 特性，3D 游戏涉及的数据可能非常多。例如，典型的 3D 游戏涉及大量的如下几种数据：

- 物体的 3D 网格；
- 2D 纹理图和光照图（light map）；
- 3D 世界；
- 运动和动画数据；
- 游戏地图数据。

在大多数情况下，占据大部分内存的数据是纹理图和光照图，因为游戏涉及的纹理图即使不是数以千计，也是数以百计。而当前使用的纹理图的颜色为 24 位，分别率高达 1024×1024，因此每个纹理图达到 3MB。当然，很多游戏允许用户选择纹理图的大小和质量，但即使用户选择的颜色为 1 位，分辨率为 256×256，每个纹理仍有 65536 字节，这可是不少的内存，另外管理内存也需要开销。在本书中，处于简化的目的（同时也是为了提高速度，因为我们使用的是软件算法），纹理的分别率将被限制到 32×32 到 128×128 的范围内；但读者应该对存储纹理需要大量的内存有所认识。另外，不但纹理需要占据内存，网格本身也要使用内存。

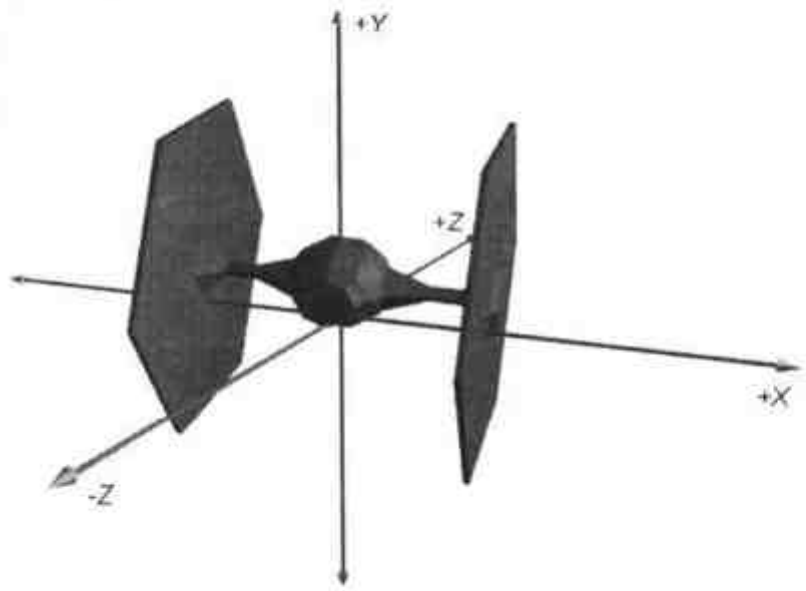


图 6.11　一个多边形模型

例如，如果使用预先录制好的动画化网格，而不是运动数据来控制动画。存储物体的网格动画所需的内存空间将急剧增加。例如，假设有一个如图 6.11 所示的物体，它由 200～300 个多边形组成，包含 200 个顶点。假设使用下面的简单数据结构来存储 3D 顶点（它能够存储位置坐标以及纹理信息和光照信息），我们来算算需要多少存储空间：

```
typedef VERTEX3Dptl_TYP
{
float x,y,z;    // 位置
float u,v;      // 纹理坐标
float l;        // 光照值（light value）

} VERTEX3Dptl, * VERTEX3Dptl_PTR;
```

每个顶点大约需要 24（6*sizeof（float））个字节（至少在 32 位系统上是这样的）。物体有 200 个顶点，因此：

24 字节/顶点 * 200 个顶点 = 480 字节/网格

然而，我们未考虑诸如多边形网格、颜色等更高级的数据。现在暂时不管它们，继续前面的例子。假设在动画中，每种运动需要 32 帧，且需要下述几种动作：

- 行走；
- 奔跑；
- 跳跃；
- 躲闪；
- 死亡；
- 射击。

因此总共需要 192（6*32）个运动帧。

而每个帧需要 480 个字节，因此总共需要 92160（192*480）字节。

这可是不少的内存。假设游戏中有 32 种动物（creature），顶点将总共占据大约 3MB（32*92160）内存。至此，读者对这些数据总共占据的内存量有所认识了吧。通用的原则是，在 3D 游戏中，必须定义高效的数据结构和数据库，并根据需要加载和卸载数据。当然，Windows 的虚拟内存管理器可提供很大的帮助，但不能同时加载和使用所有的数据，这样做将耗尽内存。

提示：另外，还必须考虑带宽。使用纹理图和大型网格时，仅仅移动物体就将占用系统总线上的大量内存带宽。这意味着高效的数据加载顺序至关重要，即加载数据后，应执行尽可能多的需要这种数据的操作。您不

希望加载并使用对象 A 后，将其丢弃并使用对象 B，然后再次加载对象 A。您希望按这样的顺序加载数据：确保缓存连贯性（cache coherence）非常高，这意味着在加载其他数据之前，尽可能多地使用当前数据。

总之，3D 游戏只不过是一个字节流，但字节流的表示方法确实至关重要；这适用于任何游戏。因此在考虑编写代码之前，创建良好的数据结构和制定深思熟虑的规划总是一个不错的主意。接下来进入 3D 部分。

6.3　3D 坐标系

前面概要性地介绍了 3D 游戏编程，接下来讨论一些 3D 内容，首先从 3D 坐标系开始。

虽然我们介绍过迪卡尔坐标（2D/3D）、柱面坐标（3D）、球面坐标（3D），但它们常用于表示 3D 系统本身。我们现在感兴趣的是，3D 图形引擎在各个处理阶段如何表示 3D 物体。

一般而言，物体被渲染到视频屏幕上之前，其坐标可能被变换到很多坐标系中，这种过程被称为 3D 观察流水线（viewing pipeline）：

- 模型坐标；
- 世界坐标；
- 相机坐标；
- 透视坐标；
- 屏幕坐标。

这些是主要的坐标，但根据具体的 3D 引擎，可能还需要经过其他类型的变换，如归一化变换和变形（warpping）变换。尽管如此，就现在而言，我们只讨论上面列出的坐标。

6.3.1　模型（局部）坐标

模型坐标（也叫局部坐标，本书将交替使用这两个术语）是 3D 实体在其局部坐标系中的坐标。也就是说，每当您创建 3D 物体时，它通常有自己的一组局部轴，物体的中心位于局部坐标系的原点。图 6.12 是一个在左手坐标系中定义的立方体。该立方体的边长为 10 个单位，中心坐标为(0，0，0)，各个顶点的坐标如下：

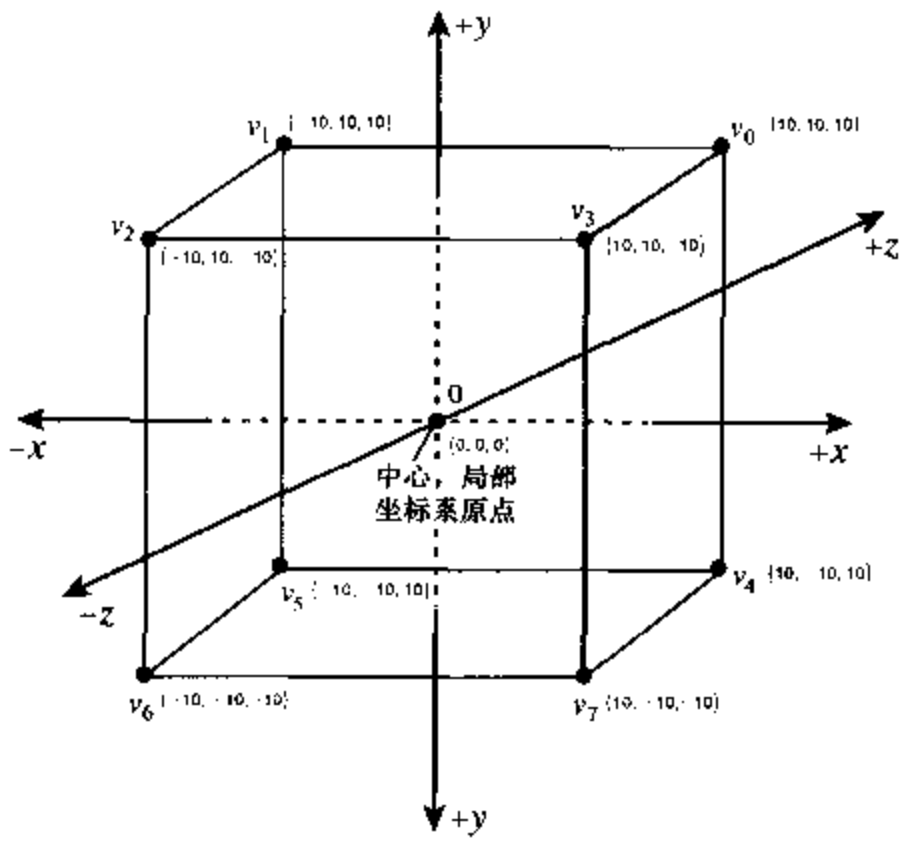


图 6.12　用模型/局部坐标表示的基本立方体

```
Vertex 0: (10,10,10)
Vertex 1: (-10,10,10)
Vertex 2: (-10,10,-10)
Vertex 3: (10,10,-10)
```

```
Vertex 4: (10,-10,10)
Vertex 5: (-10,-10,10)
Vertex 6: (-10,-10,-10)
Vertex 7: (10,-10,-10)
```

使用模型坐标的目的在于，在 3D 流水线的某些阶段，可以将模型平移到它们在 3D 世界中的实际位置。这也是 3D 游戏的工作原理。在另一方面，有时候，您不希望 3D 物体的局部坐标系原点为其几何中心。例如，假设要创建一个如图 6.13 所示的机器人手臂模型，可以将每个链杆定义为一个多边形网格，并将其中心作为局部坐标系的原点。每当物体被变换为世界坐标时，其中心将被移到相应的位置。然而，如果要旋转链杆该如何办呢？也就是说，如果要将第一个链杆作为肩膀，将第二个链杆作为前臂该如何办呢？并没有规定必须将物体的中心作为其局部坐标系的原点。对于图 6.13 所示的模型，可以将其某一端作为原点。对于这种物体，这样做更合适，因为这样原点将是旋转中心，旋转的链杆就像是一条 3D 线段。

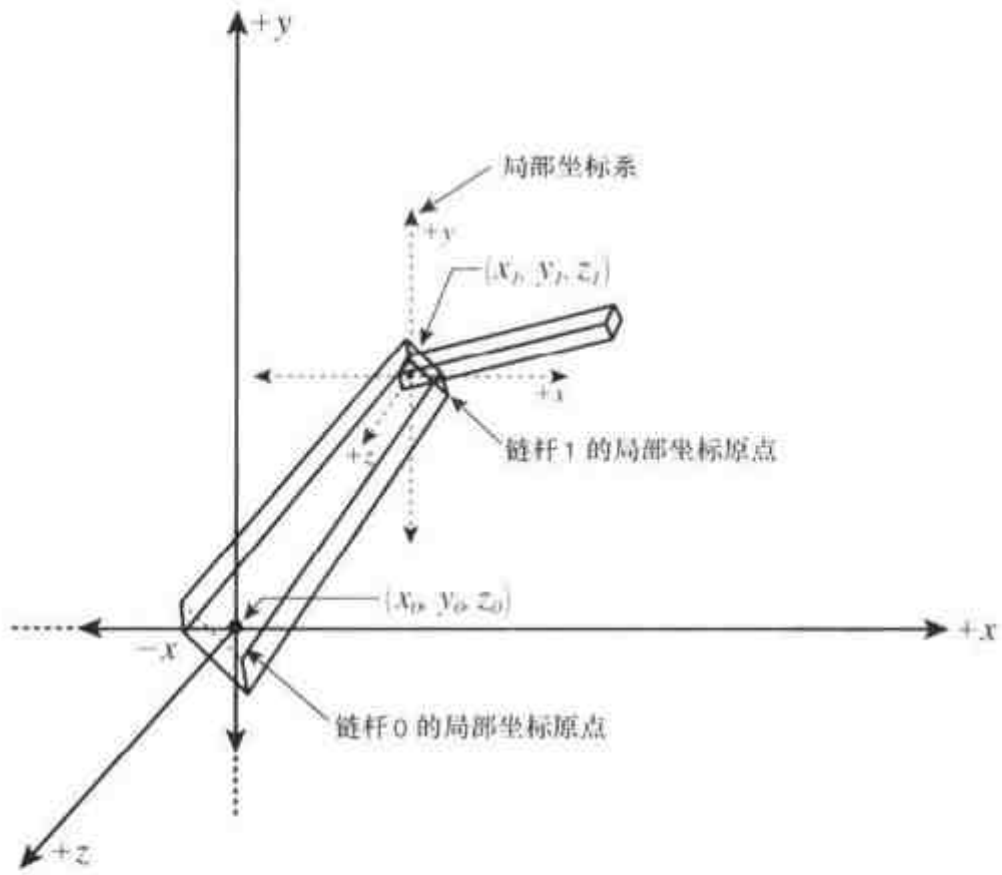


图 6.13　由多个链杆组成的机器人手臂

1. 朝向问题

最后，作者想讨论一下模型坐标系下的朝向问题。朝向指的是必须有某种定义（加载）3D 模型的约定，否则将无法确定前面或上面是哪一向。例如，假设您在 3D 游戏中定义了如图 6.14 所示的两个物体。其中宇宙飞船的前面指向负 z 轴，上面为正 y 轴；而机器人的前面为正 z 轴。您发现了其中的问题吗？

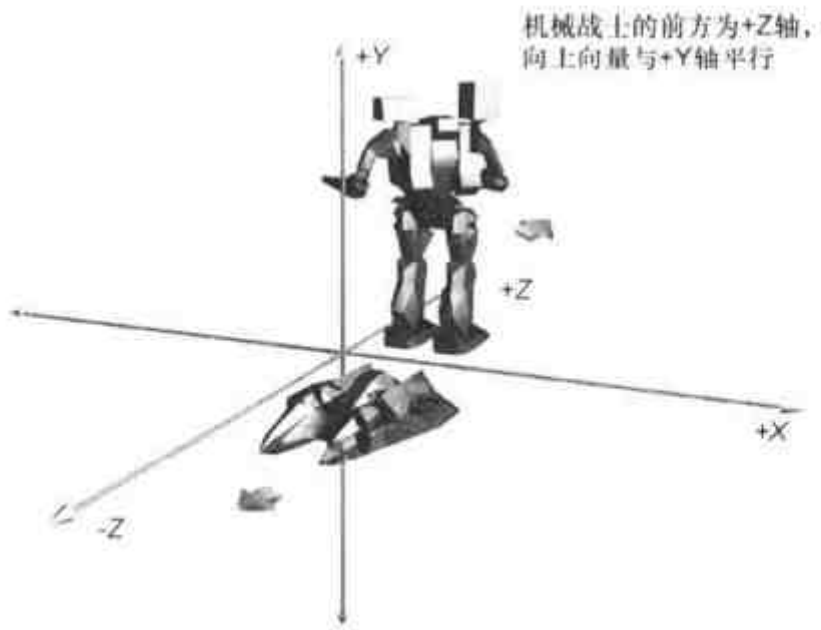


图 6.14　使用统一的朝向约定

对于朝向问题，处理方式有两种。第一种方法是，在建模阶段采用一种约定，规定所有模型的前方都必须是+z 轴，上方为+y 轴（也可以采用其他约定，只需统一即可）。第二种方法是，采用自己喜欢的方式建立物体模型，但让软件计算每个物体的主轴（最长的轴），并使之与+z 轴平行，然后计算第二个主轴，并使之与+y 轴平行。这样，不管加载时物体的朝向如何，物体的朝向都是合理的。作者的建议是，使用一种建模约定，而不要让软件去处理这项工作，因为软件只能考虑几何形状，而无法知道物体实际上看起来像什么。

在有些 2000 年以后推出的新型汽车（尤其是新型雷鸟）中，前脸看起来像后面。加载这种汽车模型时，

很容易弄错朝向，导致驾驶汽车时实际上是在倒车。

提示：即使在建模时遵循了某种朝向约定，在加载物体时也可能对其进行旋转、缩放或平移。例如，可能有一个名为 Load_3D_Mesh()的函数，它加载物体，并允许您对物体进行变换。传递给系统的模型坐标将是变换后的坐标。

2. 比例问题

模型的比例也是一个问题。在图 6.15 中，一个模型表示一辆车轮半径为 100 的汽车，另一个模型表示一辆车轮半径也为 100 的摩托车。这不合理，因为它们的缩放比例都为 1:1，这是不正确的。对于这种问题，一种解决方案是，在建模时使用相同的比例，这可能意味着建模程序中的 1 个单位总是相当于游戏引擎中的 1cm。另一种解决方案是，建立每个物体模型时，都将模型坐标空间范围设置为 1×1×1（归一化坐标），然后在加载模型时对其进行缩放。就个人而言，我总是使用相同的比例来创建所有的模型，这样无论 1 个单位表示 1cm、1m 还是 1km，模型的大小都相同。同样，加载函数可能允许您指定 *X-Y-Z* 缩放因子和 *X-Y-Z* 旋转因子，并相应地修改加载后的模型。对于诸如行星等简单物体，这项功能很有用。您可以加载同一个模型，但动态地对其缩放和旋转，这样加载后的模型看起来完全不同，从而简化了 3D 建模工作。

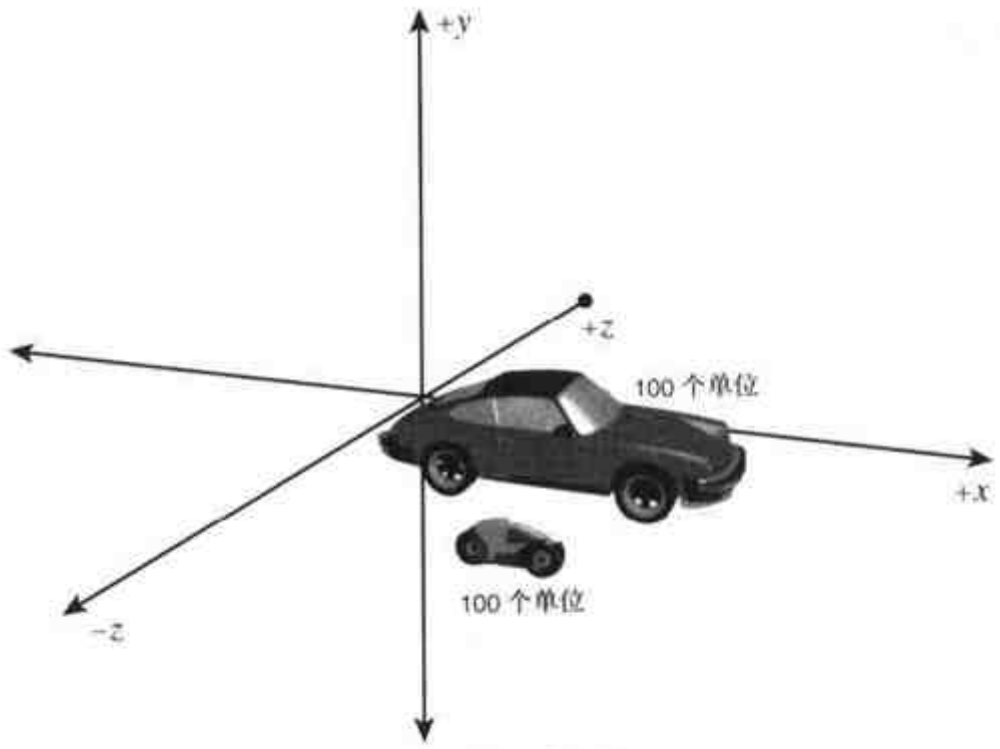


图 6.15　比例需要统一

3. LHS/RHS 坐标系

在 3D 游戏中，定义物体时，使用左手坐标系（LHS）还是右手坐标系（RHS）完全取决于您。当然，必须保持一致。作者更偏爱左手坐标系，因为这样其他的变换（和投影）将更容易，作者喜欢+z 轴垂直于屏幕朝内，而不是朝外。然而，在图纸上绘图时，有时候使用 RHS 将更容易，因为这样+z 轴将指向绘图者。要使用 RHS 模型坐标系，并在最后的 3D 投影中使用 LSH 坐标系，只需反射 *z* 轴便可实现。但为何要将简单问题复杂化呢？

6.3.2　世界坐标

建立好所有的 3D 模型后，需要将其放置到世界坐标系中。此时世界坐标将登上舞台。世界坐标表示的是虚拟宇宙（virtual universe）中的实际位置，物体将在虚拟宇宙中移动和被变换。也就是说，世界坐标表示的是绝对位置，而不是相对位置。要理解世界坐标，首先要理解其范围。即世界应多大？应为 1×1×1、10×10×10 还是 1 000 000×1 000 000×1 000 000?

如果在模型坐标中，1 个单位表示 1 米，同时希望游戏空间为 100×100×100km（对坦克和赛车而言，这很合适）。由于 1km 为 1000 米，因此游戏空间的坐标范围如下：

- x 轴—— −50 000 <= x <= 50 000；
- y 轴—— −50 000 <= y <= 50 000；

- z 轴—— −50 000 <= z <= 50 000。

对于上述范围，使用浮点数很容易对其进行处理。在另一方面，如果模型坐标的 1 个单位表示 1cm，而宇宙空间是一个 1 000 000km 的立方体，则处理世界坐标时可能遇到问题，因为世界坐标范围如下：

- x 轴—— $-0.5 \times 10^{10} <= x <= 0.5 \times 10^{10}$；
- y 轴—— $-0.5 \times 10^{10} <= y <= 0.5 \times 10^{10}$；
- z 轴—— $-0.5 \times 10^{10} <= z <= 0.5 \times 10^{10}$。

上述范围无法用 32 位整数（可表示的范围为 $\pm 2\times 10^9$）来表示，即使使用标准的 32 位浮点数来表示，也无法包含小数部分。总之，坐标范围太大了。经验法则是，需要对宇宙空间进行限制，使得即使在表示宇宙边界的坐标时，也能达到小数部分至少有 3～4 位的精度。

注意：除模型大小外，还需考虑最终的缩放变换和相机变换等因素，因此经常需要调整宇宙空间的大小，使之与游戏物体的大小、速度和缩放比例匹配。例如，在作者编写的一个 3D 射击游戏中，使用的宇宙空间非常小（4096 × 4096 × 4096），但玩家以 30 英尺/秒的速度从一端跑到另一端时，需要大约 15 秒钟，因此比较合适。然而对于空战游戏，宇宙空间需要大 10 倍，因为宇宙飞船的速度快得多。当然，实际上，宇宙飞船的速度要快数千倍，但在虚拟世界中，速度快 10 倍、空间大 10 倍可能足够了。

将物体放置到世界坐标系中

读者对局部坐标和世界坐标有所了解后，接下来介绍如何将局部坐标变换为世界坐标。别忘了，局部（模型）坐标只是用于定义物体，接下来必须将物体放置到世界坐标系中，以获得世界坐标。例如，回过头来看看图 6.12，在这个图中，用模型坐标定义了一个简单的立方体，其边长为 10 个单位。各个顶点的模型坐标如下：

```
Vertex 0: (10,10,10)
Vertex 1: (-10,10,10)
Vertex 2: (-10,10,-10)
Vertex 3: (10,10,-10)
Vertex 4: (10,-10,10)
Vertex 5: (-10,-10,10)
Vertex 6: (-10,-10,-10)
Vertex 7: (10,-10,-10)
```

实际上，我们将使用数据结构 POINT3D 来定义这个立方体：

```
POINT3D cube_model[8] = {
{10,10,10}, // 顶点 0
{-10,10,10}, // 顶点 1
{-10,10,-10}, // 顶点 2
{10,10,-10}, // 顶点 3
{10,-10,10}, // 顶点 4
{-10,-10,10}, // 顶点 5
{-10,-10,-10}, // 顶点 6
{10,-10,-10}, // 顶点 7
};
```

当然，也可以使用 4 个顶点来定义每个面，这些面是构成 3D 立方体的多边形：

```
Face 0: (0,1,2,3)
Face 1: (4,7,6,5)
Face 2: (0,3,7,4)
Face 3: (2,3,7,6)
Face 4: (1,5,6,2)
Face 5: (0,4,5,1)
```

然而，将局部/模型坐标变换为世界坐标时，并不需要多边形的定义，因此我们并不需要上述面列表。

给定用局部/模型坐标表示的物体顶点列表，要将其变换为世界坐标，只需知道要将物体的中心放置到游戏世界的什么位置，然后将局部/模型坐标平移到这个位置即可。假设游戏世界的大小为 1000×1000×1000，且要将物体的中心放置到世界坐标(world_x，world_y，world_z)上，如图 6.16 所示，则可以使用下述平移算法：

```
POINT3D cube_world[8]; // 用于存储世界坐标
```

```
// 对每个顶点平移(world_x, world_y, world_z)
for (int vertex = 0; vertex<8; vertex++)
    {
    cube_world[vertex].x=cube_model[vertex].x+world_x;
    cube_world[vertex].y=cube_model[vertex].y+world_y;
    cube_world[vertex].z=cube_model[vertex].z+world_z;
    } // end for
```

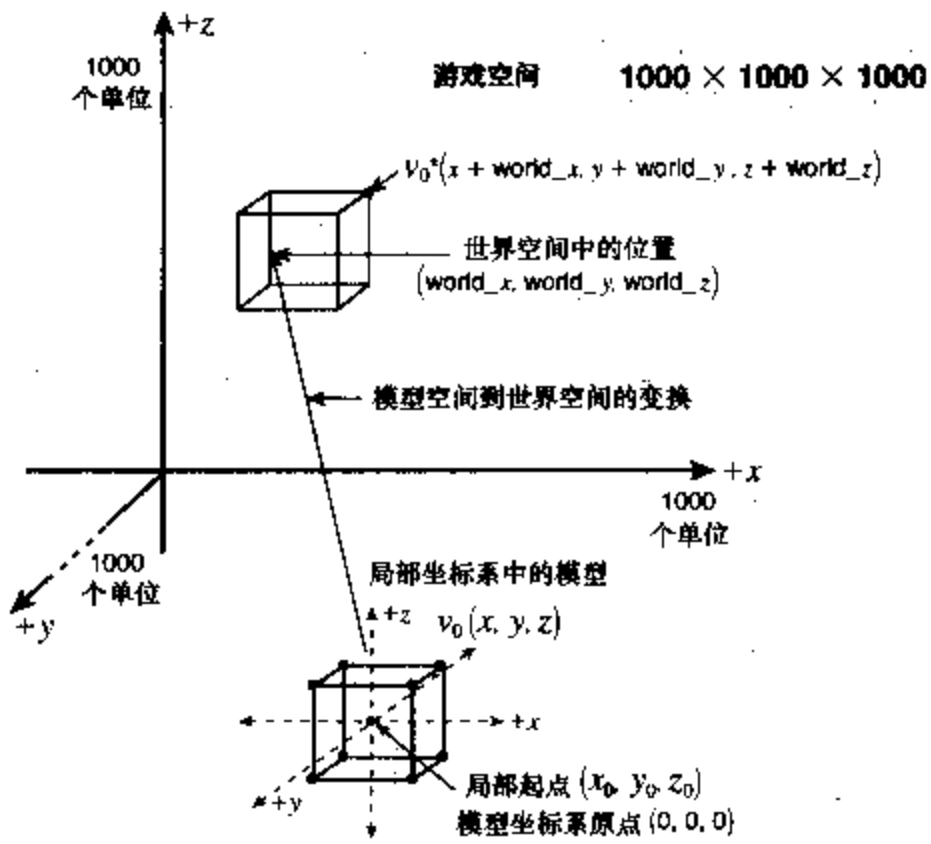


图 6.16　将物体放置到世界空间中

这里没有修改立方体的局部/模型坐标，这很重要。如果对其进行修改，将丢失立方体模型。这里对模型坐标进行变换，并将结果存储在一个新的顶点列表中。这看似浪费存储空间，但这是不可避免的。事实上，在 3D 流水线中对顶点进行处理时，可能存储多个坐标版本。我们将尽可能减少坐标版本数，但通常至少有 2～3 个版本。

上述算法只不过是进行平移而已。因此，从局部/模型坐标到世界坐标的变换可以用一个标准矩阵来表示，对于平移，这样的矩阵被称为 T_{mw}。然而，需要注意的是，要使用矩阵来表示平移，需要使用 4D 齐次坐标，平移矩阵如下：

$$T_{mw} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ world\_x & world\_y & world\_z & 1 \end{bmatrix}$$

注意：本书常常用 T 表示矩阵，用下标表示变换方式，例如 T_{lw} 表示从局部坐标到世界坐标的变换矩阵。然而，如果矩阵为平移矩阵或旋转矩阵，则 T 表示平移，R 表示旋转。总之，读者看到用 T 和下标表示的矩阵时，它要么表示常规变换，要么表示平移。

然而，本书前面介绍了很多矩阵乘法函数，它们知道如何将 3D 向量和 4×3 矩阵相乘，或者将 4D 向量同 4×3 和 4×4 矩阵相乘。关键之处在于，要使用矩阵乘法来实现平移，必须使用包含 4 行的矩阵。要将点[x_m y_m z_m 1]从局部/模型坐标变换为世界坐标，需要执行的矩阵运算如下：

$$[x_m\ y_m\ z_m\ 1] * T_{mw}$$

$$= [x_m\ y_m\ z_m\ 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ world\_x & world\_y & world\_z & 1 \end{bmatrix}$$

$$= [x_m + world\_z\ y_m + worldy\ z_m + world\_z\ 1]$$

$$= [x_w\ y_w\ z_w\ 1]$$

为简化数学运算，这里使用的是 4D 齐次坐标，但只要将第 4 个分量 w 视为 1.0，也可以使用 3D 坐标，因此可以根据情况，分别使用 4×3 或 4×4 矩阵乘法。最后，要将 4D 齐次坐标转换为常规的 3D 坐标，只需将 x_w、y_w 和 z_w 分别除以分量 w 即可。在这个例子中，w 为 1.0，因此无需执行除法。

总之，将局部/模型坐标变换为世界坐标很简单，只需将每个物体的中心平移到世界空间中所需的位置即可。这是通过将模型的所有顶点进行平移，并将结果存储到另一个数据结构中实现的。

在 3D 流水线中，仅当需要旋转或缩放物体时，才需要对物体的局部坐标进行变换；否则只需将局部/模型坐标变换为世界坐标，从而将物体放置到世界空间中。图 6.17 是至此完成的 3D 流水线，其输出为世界坐标。

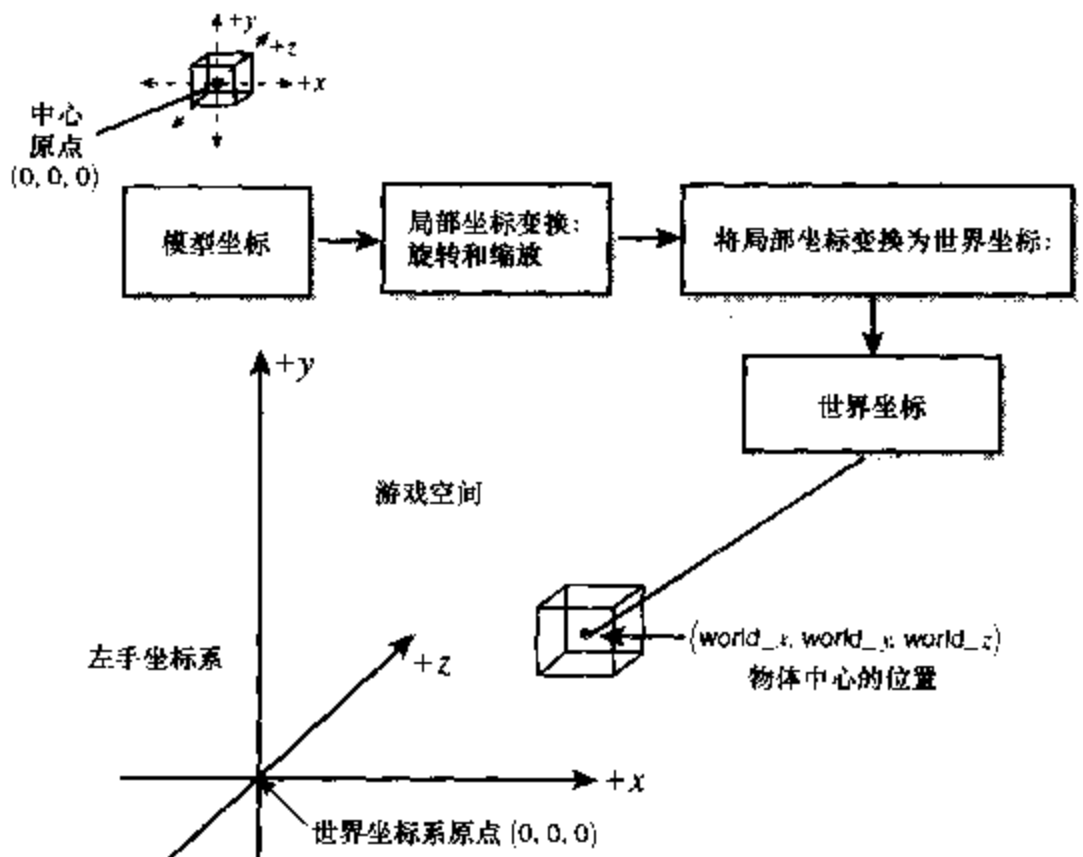


图 6.17　包含局部坐标到世界坐标变换的 3D 流水线

6.3.3　相机坐标

局部/模型坐标在物体自己的 3D 空间内定义了物体，坐标系是相对于(0，0，0)的；世界坐标系定义了虚拟的 3D 空间，物体将被放置到这个空间中。然而，3D 游戏是围绕着 3D 相机进行的，因此需要采取某种方式来观察 3D 物体。为此，需要定义相机坐标系，并将世界坐标变换为相机坐标。

1．视景体

要观察 3D 空间中的场景，最常用的方法是将一台相机放置在世界的某个地方，然后通过特定的视景体观察世界，如图 6.18 所示。正如读者看到的，有一个 3D 世界坐标系，一台虚拟相机放置在(cam_x，cam_y，cam_z)处，相机的方向是由(ang_x，ang_y，ang_z)定义的。另外，还有一个视景体，它定义了相机能够拍摄到的空间。视景体可以通过水平和垂直方向的视野（FOV）参数来定义的，这两个参数的取值范围通常为 60～130 度。另外，还有远裁剪面和近裁剪面，只有位于这两个裁剪面之间的物体才是可见的，其他物体都看不到。最后，还有更复杂的视平面，但归根结底，它是一个数学意义上的平面，3D 图像将被投影到该平面上，以便能够在 2D 计算机屏幕上渲染图像。接下来我们花些时间来讨论构成视景体的 6 个面。

（1）远、近裁剪面

远裁剪面和近裁剪面分别位于 far_z 和 near_z 处。它们的平面方程分别是 z = near_z 和 z = far_z。

这些平面垂直于观察方向，决定了哪些物体将被渲染到图像中。换句话说，比远裁剪面更远或比近裁剪面更近的物体都不会被渲染。比远裁剪面更远的物体被投影和渲染后将非常小（小点），因此没必要考虑它们；相反，比近裁剪面更近的物体离相机太近，如果考虑它们，将相当于让您面向一堵墙，并看着它。

（2）视景体墙

视景体实际上是使用远裁剪面和近裁剪面对一个棱锥进行裁截得到的，该棱锥是由视景体墙定义的。

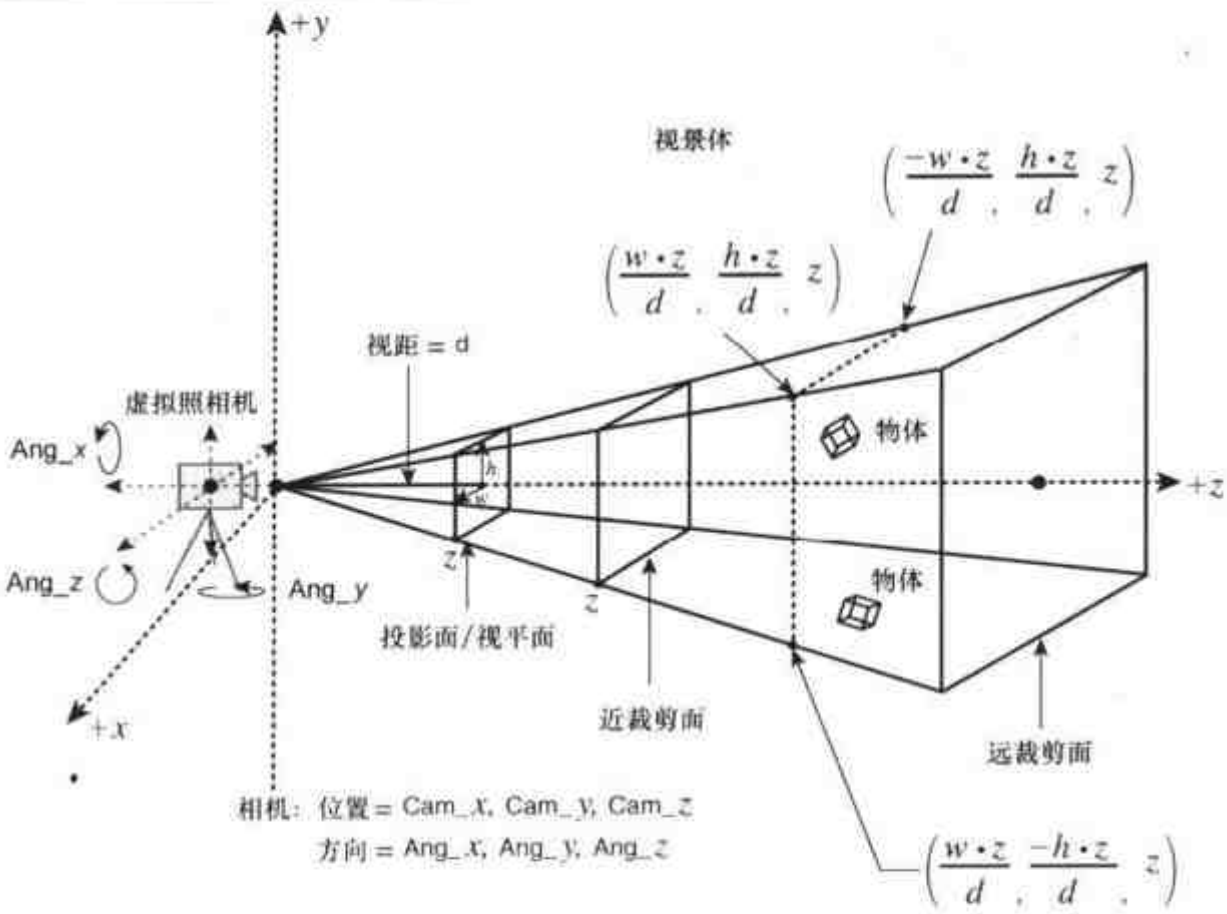


图 6.18　3D 视景体

在图 6.18 中，使用的是一个 90 度的视景体，四面视景体与观察方向的夹角都为 45 度，如图 6.19 所示。因此，视景体墙平面的方程为$|x| = z$ 和$|y| = z$。和远近裁剪面一样，视景体墙也被用于裁剪物体。不再视景体墙内面的物体是看不到的，将被裁减掉（删除）。

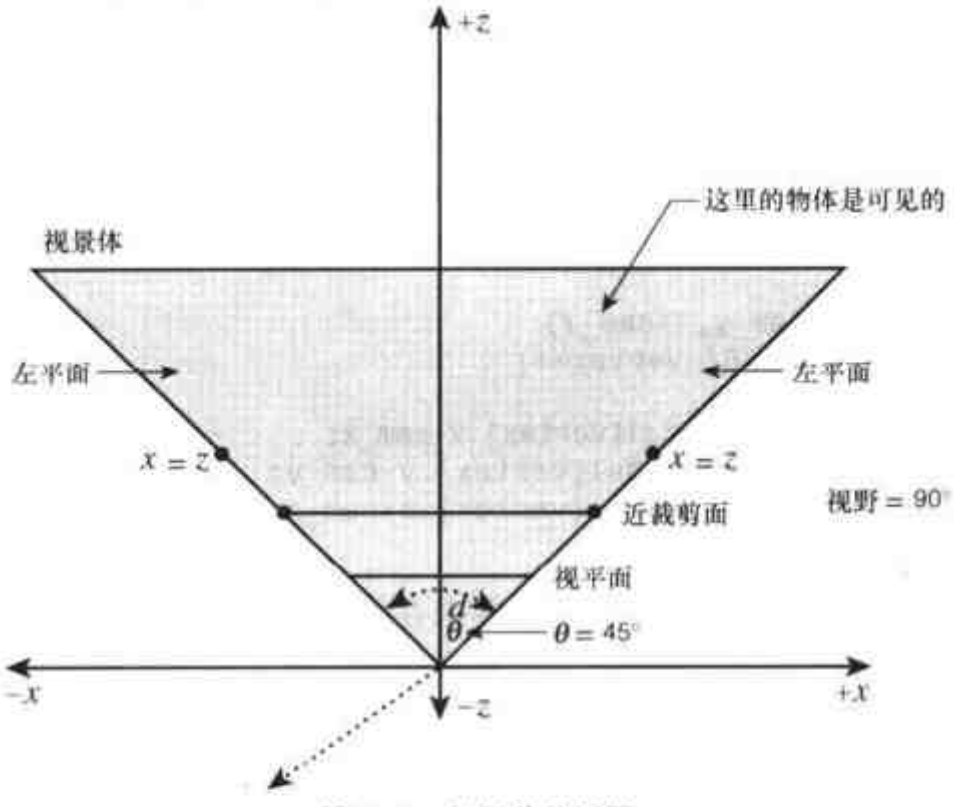


图 6.19　视景体俯视图

（3）世界和相机之间的关系

我们要对 3D 世界中的物体进行变换，使其位置是相对于相机的。换句话说，我们要以尽可能简单的方式将物体的世界坐标同相机关联起来，即需要将世界坐标变换为相机坐标。

在图 6.19 中，相机的世界坐标为(0，0，0)，镜头指向+z 轴方向，水平视野和垂直视野都是 90 度。如果物体位于视景体内，将被渲染；否则不被渲染。然而，在 99.99999%的情况下，相机都不位于原点处，也不是指向+Z 轴方向，而是位于玩家的眼睛处，并指向玩家前面。

如果要人为地世界坐标系，使相机位于世界坐标系的原点处，指向+z 轴方向，且上方为+y 轴方向，即观察角度为(0，0，0)，需要对所有的世界坐标进行什么样的变换呢？这就是世界坐标到相机坐标变换，实际上非常简单。

2. 世界坐标到相机坐标的变换

世界坐标到相机坐标的变换指的是对世界坐标系进行移动，使得相机位于世界坐标系的原点，镜头指向+Z 轴方向；为此，只需相应的移动所有物体即可。这种变换由两步组成：首先平移，然后旋转。我们首先讨论平移。

（1）平移

图 6.20a 是一个经过简化的游戏世界，其中只有一个物体和一台相机，后者的位置为(cam_x，cam_y，cam_z)。为简化问题，假设相机角度为(0，0，0)。如果将相机移到世界坐标(0，0，0)处，要确保位于(world_x，world_y，world_z)的物体相对于相机的位置保持不变，应如何移动该物体呢？在图 6.20b 中，相机移到了(0，0，0)处，为保持物体和相机的相对位置不变，物体被移到(world_x－cam_x，world_y－cam_y，world_z－cam_z)处。相机和物体之间的相对距离和角度保持不变；换句话说，相机镜头中的风景依旧。

相机镜头指向+z 轴后，可以通过一些数学运算来判断物体是否在视景体内，如果是，则再通过其他一些数学计算将其渲染到屏幕上。这些内容将在后面介绍。关键之处在于，我们已经解决了世界坐标到相机坐标变换这一问题的一半——平移。

换句话说，世界坐标到相机坐标变换的第一步是对所有世界坐标进行平移，平移量为相机位置的世界坐标的负，即(－cam_x，－cam_y，－cam_z)。完成这一步的代码如下（这里使用的是前一节中的世界坐标）：

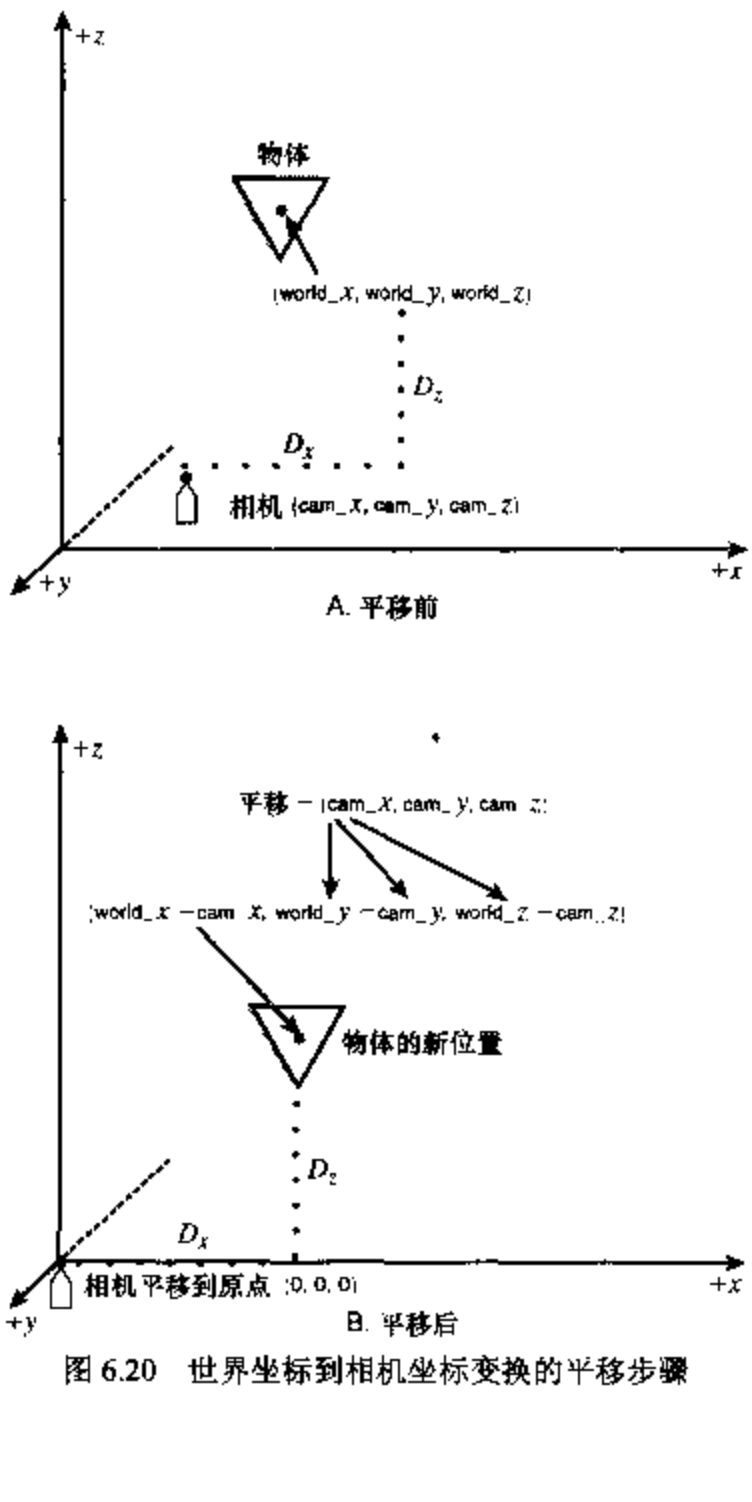


图 6.20　世界坐标到相机坐标变换的平移步骤

```
POINT3D cube_camera[8]; // 用于存储相机坐标

// 将每个顶点平移(-cam_x, -cam_y, -cam_z)
for (int vertex = 0; vertex<8; vertex++)
   {
    cube_camera[vertex].x=cube_model[vertex].x-cam_x;
    cube_camera[vertex].y=cube_model[vertex].y-cam_y;
    cube_camera[vertex].z=cube_model[vertex].z-cam_z;
    } // end for
```

同样，这里使用了一个新的 POINT3D 结构数据来存储相机坐标。另外，这一步类似于局部坐标到世界坐标变换，读者可能想将其合并成一次变换。虽然现在只完成了世界坐标到相机坐标变换工作的一半，但我还是要说，可以使用矩阵乘法将 3D 流水线中的很多变换合并起来。然而，很多时候，最好还是分步完成这些工作，因为我们可能不想将所有物体的世界坐标都变换为相机坐标。在 3D 流水线处理过程中，很多由顶点定义的物体和/或多边形可能由于诸如裁剪和隐藏面消除等操作而被丢弃。然而，根据 3D 引擎的工作方式，确实可能使用一个矩阵将点的局部坐标直接变换为屏幕坐标。这将在后面介绍。

同样，可以使用一个矩阵来表示上述平移步骤。假设相机位置的平移矩阵为 T_{cam}，则将其每个平移因子求负，可得到其逆矩阵 T_{cam}^{-1}：

$$T_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

数学知识：对于任何平移矩阵，只需将其最下面一行中的平移因子求负，便可以得到其逆矩阵。另外，对于

任何旋转矩阵，要计算其逆矩阵，只需将该矩阵中每个正弦和余弦项中的角度求负，或将其转置。

因此，要计算相机被人为地移动到原点后，各个顶点的相应位置，只需将其世界坐标与矩阵 T_{cam}^{-1} 相乘即可。具体步骤如下（这里没有考虑相机的方向）：

$$[x_w\ y_w\ z_w\ 1] * T_{cam}^{-1}$$

$$= [x_w\ y_w\ z_w\ 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

$$= [x_w - cam\_x\ y_w, -cam\_y\ z_w, -cam\_z, 1]$$

如果要在代码中使用矩阵来完成这个步骤，需要使用简单的矩阵乘法，如下所示：

```
// 定义平移矩阵
MATRIX4X4 Tcam_inv = { {1, 0, 0, 0},
                       {0, 1, 0, 0},
                       {0, 0, 1, 0},
                       {-cam_x, -cam_y, -cam_z, 1} };

// 通过矩阵乘法将每个顶点平移(-cam_x, -cam_y, -cam_z)
for (int vertex = 0; vertex<8; vertex++)
    {
    // 将向量/点 cube_world[]乘以平移矩阵
    // 为确保矩阵乘法运算有效，假设 w=1
    Mat_Mul_VECTOR3D_4X4(cube_world[vertex],
                         &Tcam_inv,
                         cube_camera[vertex]);

    } // end for
```

警告：执行矩阵乘法时，务必使用不同的源顶点数组和目标顶点数组，否则结果将不正确。因此使用矩阵对顶点进行变换时，决不要将变换结果存储到原来的顶点变量中。

至此，我们完成了世界坐标到相机坐标变换的第一步，接下来需要完成第二步。在第一步中，我们首先将相机平移到坐标系的原点，而对每个物体或顶点的世界坐标执行一个逆平移操作，即平移量为(-cam_x，-cam_y，cam_z)；对于相机的朝向，也可以以类似的方法来处理。

（2）旋转

图 6.21a 几乎与图 6.20a 相同，只是相机观察角度为(0，ang_y，0)。换句话说，相机的航向角（yaw angle）不再为 0，因此不再指向+z 轴方向。如果能够确定将相机的朝向旋转到与+z 轴平行对世界中其他物体的影响，问题便解决了。

首先，需要平移所有物体的世界坐标，使相机位于世界坐标系的原点。为此，可对相机的位置坐标求负，然后将其作为所有物体的平移量，即平移(-cam_x，-cam_y，cam_z)。这相当于将每个点的世界坐标乘以矩阵 T_{cam}^{-1}。

此时的情形如图 6.21b 所示。相机位于世界坐标系原点，物体相对于相机的位置保持不变，但相机的观察角仍为(0，ang_y，0)。要让相机的朝向与+Z 轴重合，只需将相机绕 Y 轴旋转-ang_y，这是相机观察角度(0，ang_y，0)的逆。然而，我们必须对所有物体的顶点也这样做，结果如图 6.21c 所示。

在图 6.21c 中，相机位于(0，0，0)处，观察角度为(0，0，0)，同时，物体与相机之间的相对位置和角度保持不变。只有物体在世界空间的绝对位置发生了变化，但物体之间以及物体和相机之间的相对关系保持不变。至此，对于相机位置为(cam_x，cam_y，cam_z)和欧拉观察角度为(0，ang_y，0)的情况，从世界坐标到相机坐标的变换便完成了。

然而，对于一般性观察角度(ang_x，ang_y，ang_z)，又该如何处理呢？变换方法相同：必须首先使用逆矩阵 T_{cam}^{-1} 对世界坐标进行变换，然后使用相机朝向的逆进行一系列的旋转变换，将世界坐标变换为合适的值。当然，相机朝向由 3 个角度指定，因此旋转顺序有 6 种：

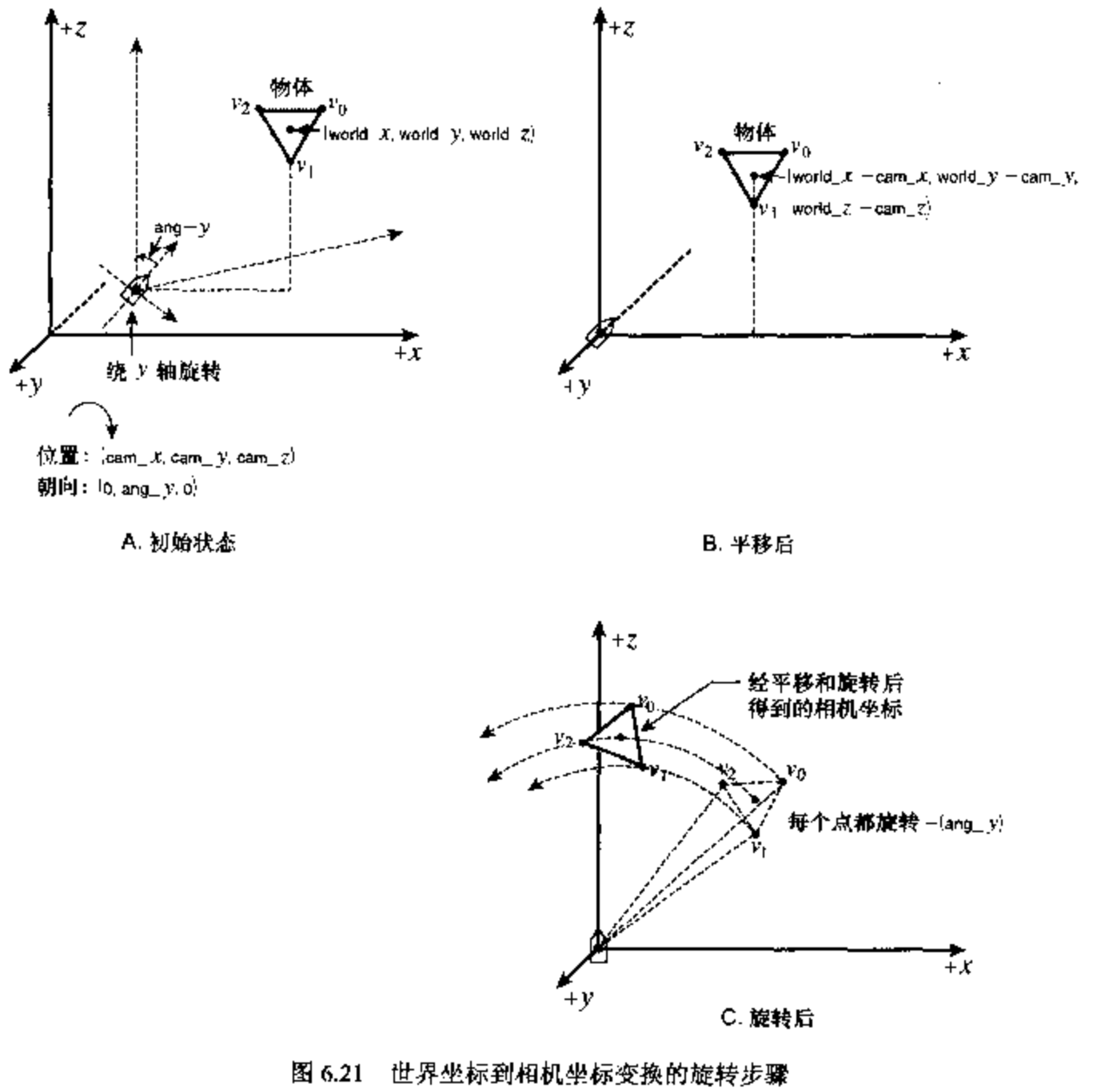


图 6.21　世界坐标到相机坐标变换的旋转步骤

- 顺序 1：xyz；
- 顺序 2：xzy；
- 顺序 3：yxz；
- 顺序 4：yzx；
- 顺序 5：zxy；
- 顺序 6：zyx。

然而，大多数人都使用顺序 3 或顺序 6，因为首先改变航向，然后改变倾斜角更为合理。除非对狗而言，否则倾侧角不容易理解。然而，采用哪种顺序完全取决于您。顺序只是意味着在有虚拟相机的情况下，按什么样顺序执行旋转，仅此而已。然而，在大部分情况下，游戏中并没有相机，而只有相机拍摄到的景色；因此旋转顺序是无关紧要的。无论如何，我们在范例中将采用顺序 3。

因此，需要执行 3 次旋转：一次绕 y 轴，我们称之为 R_{camy}^{-1}；然后绕 x 轴，称之为 R_{camx}^{-1}；再绕 z 轴旋转，称之为 R_{camz}^{-1}。另外，旋转必须是相机角度的逆，因为我们需要反转整个系统。对于每个顶点，将其变换为相机坐标的完整过程如下：

$$T_{wc} = T_{cam}^{-1} * R_{camy}^{-1} * R_{camx}^{-1} * R_{camz}^{-1}$$

当然，可以将这些变换矩阵合并成一个矩阵，称之为 T_{wc}，然后使用它对每个顶点进行变换，以便将世界坐标变换为相机坐标，如下所示：

$$[x_w \ y_w \ z_w \ 1] * T_{wc}$$

提示：注意，这里使用的是相机平移矩阵和旋转矩阵的逆矩阵。如果将相机视为一个物体，为正确地指定为位置和朝向，可使用下述变换序列将其变换为世界坐标：

$R_{camy} * R_{camx} * R_{camz} * T_{cam}$

注意，平移和旋转顺序相反，且使用的不是逆矩阵。另外，相机不过是一个单位向量而已，其位置为（cam_x，cam_y，cam_z），方向为（ang_x，ang_y，ang_z）。

在实际编程时，您需要计算矩阵 T_{wc}，它等于 $T_{cam}^{-1} * R_{camy}^{-1} * R_{camx}^{-1} * R_{camz}^{-1}$。您可以在草稿纸上手工计算出这个矩阵，然后使用它来初始化；也可以定义各个矩阵，然后依次将它们相乘，并使用得到的结果。这两种方式都可行，因此对于每一帧，只需计算世界坐标到相机坐标变换矩阵一次，且几次矩阵乘法运算的工作量并不大。

然而，如果您确实要手工计算 T_{wc}，可使用三角函数来简化矩阵乘法。然而，我想将这项内容放到后面去介绍，因为这只不过是一个三角函数练习而已。就现在而言，我们将使用前面的平移和旋转公式，让计算机完成这项工作：

```
// 假定相机位于(cam_x, cam_y, cam_z)处
// 观察角度为 (ang_x, ang_y, ang_z)
// 旋转顺序为 yxz

// 平移矩阵
MATRIX4X4 Tcam_inv = { {1, 0, 0, 0},
                       {0, 1, 0, 0},
                       {0, 0, 1, 0},
                       {-cam_x, -cam_y, -cam_z, 1} };

// 现在需要计算绕 x、y 和 z 轴旋转的旋转矩阵

// 下面是绕 x 轴的逆旋转矩阵
// 要得到逆矩阵，只需要将旋转角度取负即可
MATRIX4X4 Rcamx_inv = {
        {1,   0,           0,              0},
        {0,   cos(-ang_x), sin(-ang_x),    0},
        {0,  -sin(-ang_x), cos(-ang_x),    0},
        {0,   0,           0,              1}};

// 下面是绕 y 轴的逆旋转矩阵
// 要得到逆矩阵，只需要将旋转角度取负即可
MATRIX4X4 Rcamy_inv = {
          {cos(-ang_y), 0,  -sin(-ang_y),   0},
          {0,           1,   0,             0},
          {sin(-ang_y), 0,   cos(-ang_y),   0},
          {0,           0,   0,             1}};

// 下面是绕 z 轴的逆旋转矩阵
// 要得到逆矩阵，只需要将旋转角度取负即可
MATRIX4X4 Rcamz_inv = {
          {cos(-ang_z), sin(-ang_z),  0,   0},
          {-sin(-ang_z), cos(-ang_z), 0,   0},
          {0,            0,           1,   0}
          {0,            0,           0,   1}};

// 计算出所有的逆旋转矩阵后，将它们相乘
// Tcam_inv * Rcamy_inv * Rcamx_inv * Rcamz_inv

MATRIX4X4 Mtemp1, Mtemp2, Tcam;
Mat_Mul_4X4(&Tcam_inv, &Rcamy_inv, &Mtemp1);
Mat_Mul_4X4(&Rcamx_inv, &Rcamz_inv, &Mtemp2);

// 注意顺序
Mat_Mul_4X4(&Mtemp1, &Mtemp2, &Tcam);

// 现在可以执行世界坐标到相机坐标变换了

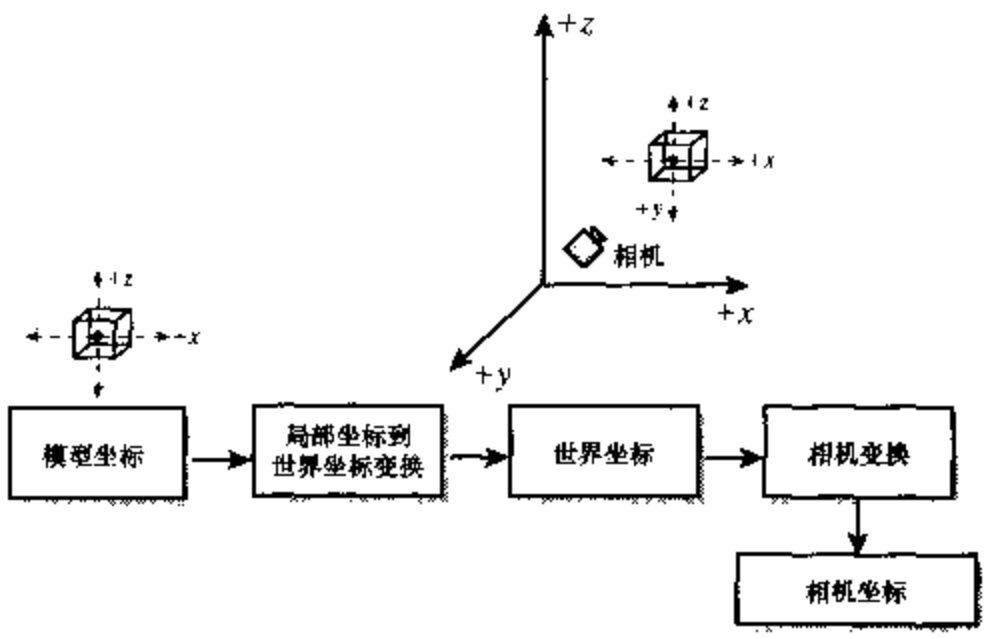
for (int vertex = 0; vertex<8; vertex++)
    {
    // 将向量/点 cube_world[]乘以平移矩阵
```

```
    // 为确保矩阵乘法运算有效，假设 w=1
    Mat_Mul_VECTOR3D_4X4(cube_world[vertex],
                         &Tcam,
                         cube_camera[vertex]);

    } // end for
```

数学知识：如果读者愿意，可以使用恒等式 sin（-x）＝-sin（x）和 cos（-x）＝cos（x）来简化矩阵中的正弦和余弦项。这里之所以没有这样简化，旨在向读者说明计算旋转矩阵的逆矩阵是多么简单。

现在，可以在 3D 观察流水线中加入世界坐标到相机坐标的变换，得到如图 6.22 所示的流程图。然而，该流程图并不准确，其中省略了一些中间步骤，如隐藏物体消除和裁剪，稍后将介绍这些步骤。



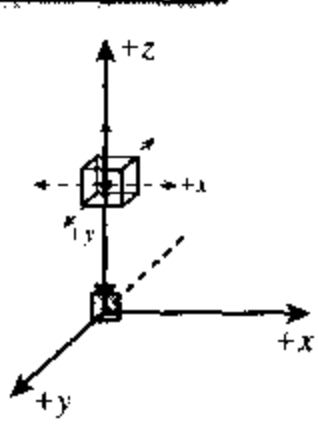


图 6.22　包含世界坐标到相机坐标变换的 D3 流水线

6.3.4　有关相机坐标的说明

最后需要提醒读者注意的是，世界坐标到相机坐标变换取决于您使用的虚拟相机表示方法。在本章中，我们使用的是一种非常粗糙的相机系统，这种系统存在很多问题，如万向接头死锁（gimbal lock）。

将相机绕某条轴旋转，使其达到与另一条轴重叠的程度后，再将相机绕后一条轴旋转时根本不管用。例如，在图 6.23 中，首先将相机绕 y 轴旋转 90 度，使其与 x 轴重叠后，再绕 X 轴旋转相机时根本不管用，这被称为万向接头死锁。

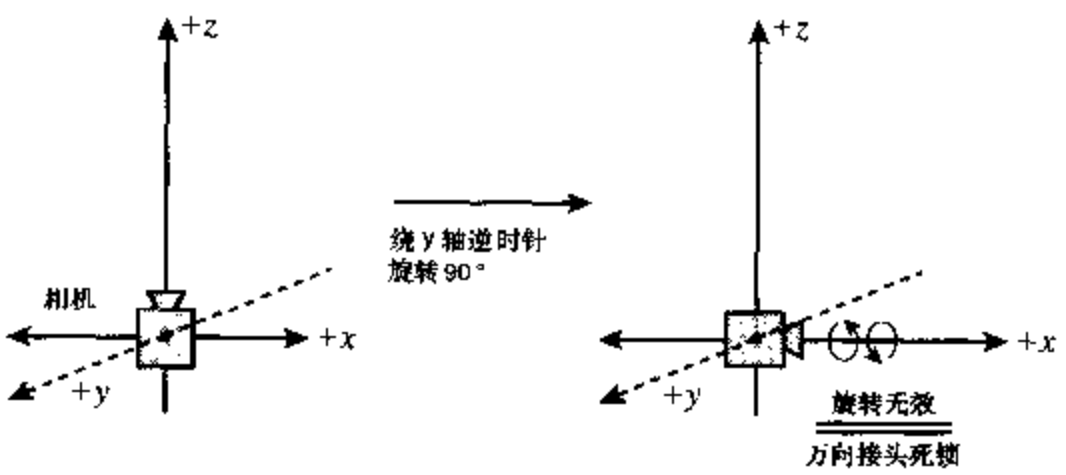


图 6.23　万向接头死锁

当然，有更好的相机系统，它们基于一种更自然的相机表示法，并从焦点（或注视点[look at point]的角度考虑相机的朝向和位置，如图 6.24 所示，这将在下一章详细介绍。

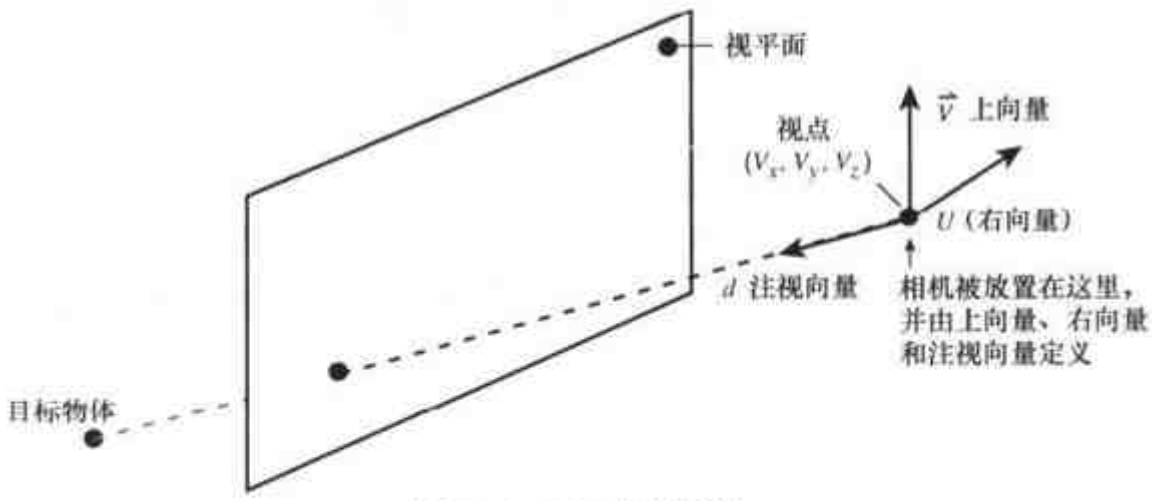


图 6.24　UVN 相机系统

6.3.5　隐藏物体（面）消除和裁剪

前面讨论坐标变换时，有两个主题——隐藏物体（面）消除和裁剪——没有介绍，因为它们与坐标没有关系。但要提醒读者注意的是，在 3D 流水线中需要考虑这两个主题。接下来花些时间来讨论它们，首先介绍隐藏物体（面）消除。

执行世界坐标到相机坐标变换时（或在此之前），需要执行很多测试，以判断物体对相机而言是否可见。这些测试通常被称为隐藏面消除。也就是说，需要判断哪些物体和/或几何体对相机而言是可见的，以免错误地渲染它们（更重要的是，首先避免对物体进行相机变换）。通常，执行的测试有两种：背面消除和包围球测试。

1．背面消除

在图 6.25 中，有一个物体和一台相机，但目前还处于世界坐标系下。隐藏面消除测试是在世界空间中进行的，测试还未执行世界坐标到相机坐标变换。这种测试用于避免对那些由于被遮住而从视点看不到的多边形进行世界坐标到相机坐标变换。

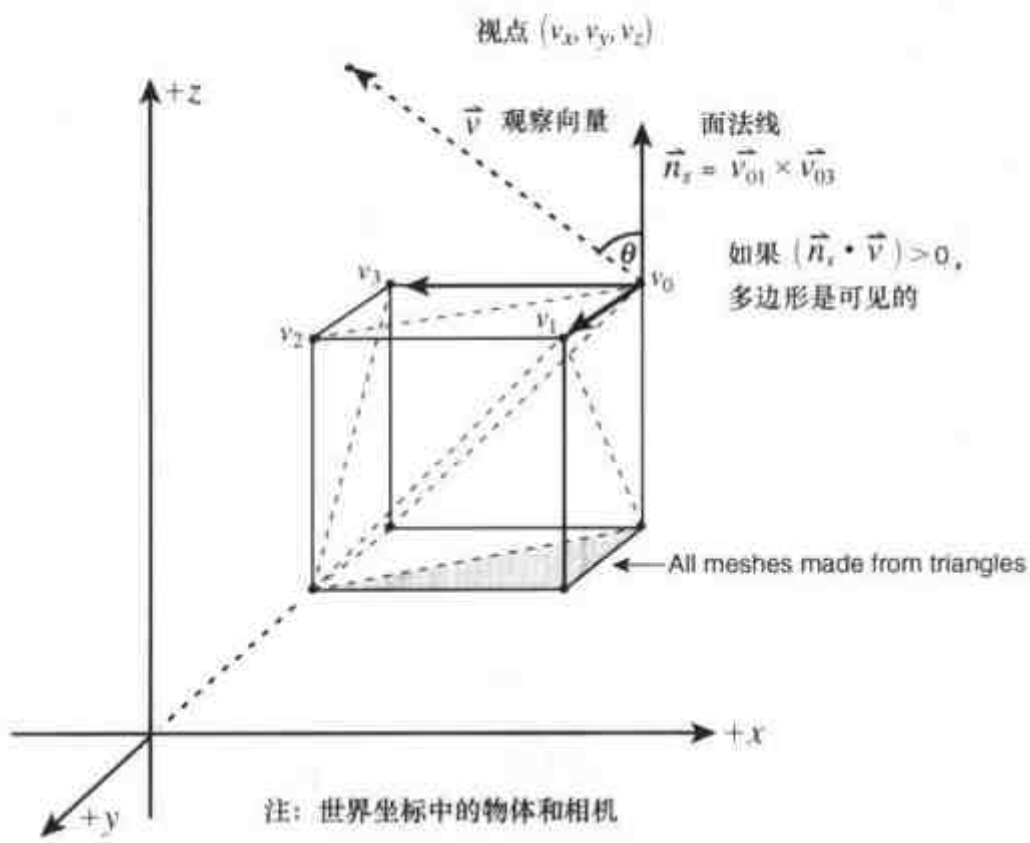


图 6.25　背面消除

背面测试的工作原理如下：以统一的方式（顺时针或逆时针，这无关紧要）对构成每个物体的所有多边形进行标记，然后计算每个多边形的面法线 $\mathbf{n}_s$，并根据观察向量对这条法线进行测试，如图 6.25 所示。

如果面法线和观察向量之间的夹角不超过 90 度，则多边形对观察者而言是可见的（当然，对于双面多边形，这种测试不管用）。使用这种测试，最起码能消除一半的多边形。

这种测试是使用点积来实现的，因为任何两个向量的点积都具有如下性质：

- 当且仅当向量 u 和 v 的夹角为 90 度（$\pi/2$ 弧度）时，$u.v=0$；
- 当且仅当向量 u 和 v 的夹角小于 90 度（$\pi/2$ 弧度）时，$u.v>0$；
- 当且仅当向量 u 和 v 的夹角大于 90 度（$\pi/2$ 弧度）时，$u.v<0$。

因此，在以多边形网格定义物体的数据结构（稍后将介绍）中，有一个指出多边形为单面的标记，这样便可以使用这种测试来排除大量的多边形，避免对它们进行开销高昂的世界坐标到相机坐标变换，伪代码如下：

```
if ( n0. v) > 0)
   {
   // 多边形是可见的
   } // end if
```

注意：您可能将面法线与观察方向的夹角刚好为 90 度（即位于视野边缘）的多边形也视为可见的。然而，这在渲染时可能带来问题，因为这些多边形的宽度通常为 1 个像素。

2. 包围球测试

执行背面消除——根据相机在世界坐标系中的位置排除尽可能多的多边形后，需要对余下的所有物体进行世界坐标到相机坐标变换。然而，在执行背面消除之前，可以执行另一种测试。事实上，应该在背面消除之前执行该测试，以排除物体。然而，仅当物体被隔离到凸形区域中时这种测试才管用。换句话说，如果游戏是在太空中进行的，可以使用框（或球面）将每个行星和宇宙飞船围住。另一方面，对于由单个网格组成的室内游戏，这种测试不适用。然而，您完全可以将房间划分成多个区域。关键之处在于，尽可能避免对游戏世界中不可见的大型区域进行处理。

这种测试的原理是，对于世界空间中的每个物体，创建一个将其包围起来的球体，如图 6.26 所示。然后，只对球心（单个点）执行世界坐标到相机坐标变换，并判断球体是否位于视景体内，如果不在视景体内，则丢弃它包围的整个物体。如果球体的一部分在视景体内，这种测试无法得出结论，需要做其他测试。

因此，如果游戏世界中的物体由多边形构成，首先需要在世界空间内计算出包围每个物体的球体，然后只对（由单个点定义的）球体进行世界坐标到相机坐标变换，并判断它是否在视景体内。这样的物体将被丢弃，即包围它的球体完全不在视景体内。接下来，执行背面消除，并对余下的多边形执行世界坐标到相机坐标变换，这些多边形不是背面，且位于视景体内。

有关包围球算法的细节，将在创建线框 3D 引擎后介绍；这里只简述一下其中涉及的数学知识。问题很简单。假设有一个物体 O，它包含一组顶点，要计算包围球，可找出哪个顶点离物体中心最远，该顶点与中心之间的距离就是包围球的半径。这种计算可在加载物体前完成，而不要实时地计算。下面是一种可行的算法，它使用的是局部/模型坐标：

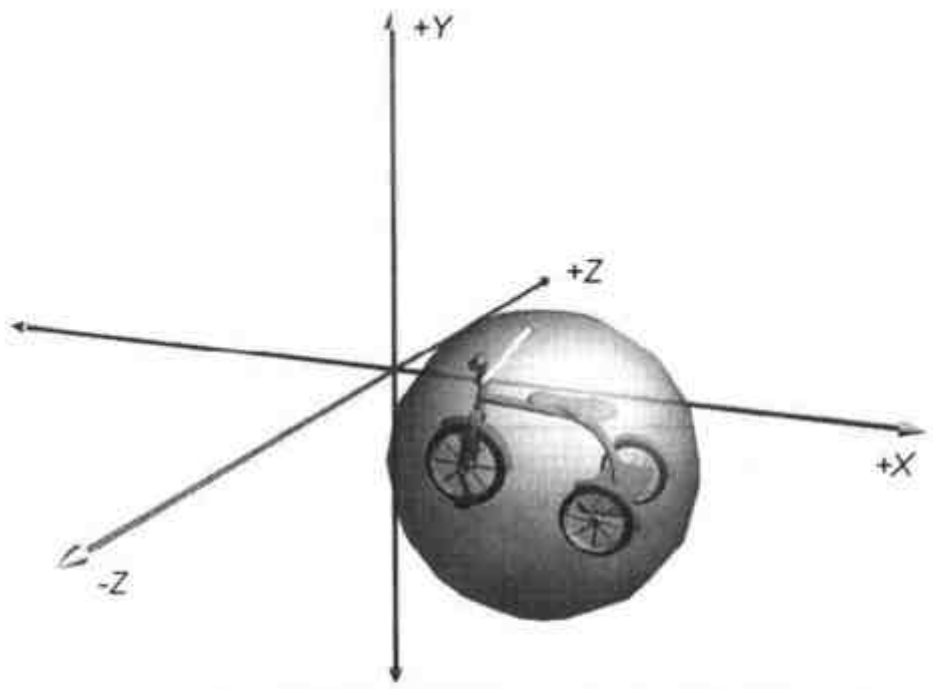


图 6.26　使用球体将物体包围起来以执行视景体剔除测试

```
POINT3D cube_model[NUM_VERTICES]; // 3D模型

float max_radius = 0; // 设置为0
float curr_radius;

// 遍历顶点列表，将curr_radius设置为最大的半径
for (int vertex=0; vertex < NUM_VERTICES; vertex++)
    {
    float x=cube_model[vertex].x;
    float y=cube_model[vertex].y;
    float z=cube_model[vertex].z;

    // 当前半径是否更大?
    if (curr_radius = sqrt(x*x + y*y + z*z) > max_radius)
        max_radius = curr_radius;

    } // end for
```

当然，这里大量地使用了函数 sqrt()。要避免这样做，可以首先找出最大的距离平方，然后调用函数 sqrt() 一次来计算出包围球半径：

```
POINT3D cube_model[NUM_VERTICES]; // 3D模型

float max_radius = 0; // 设置为0
float curr_radius;

// 遍历顶点列表，计算最大的半径平方
for (int vertex=0; vertex < NUM_VERTICES; vertex++)
    {
    float x=cube_model[vertex].x;
    float y=cube_model[vertex].y;
    float z=cube_model[vertex].z;

    // 半径平方是否更大?
    if (curr_radius = (x*x + y*y + z*z) > max_radius)
        max_radius = curr_radius;

    } // end for

// 计算半径
max_distance = sqrt(max_distance);
```

提示：上述代码基于的原理是，如果|a|>|b|，则 $a^2>b^2$，反之，如果|a| <|b|，则 $a^2<b^2$。

具体采用哪种算法无关紧要，因为这是在初始化阶段进行的。计算出半径后，可以通过一些计算来创建包围球：给定物体中心 **p0** 的世界坐标(x0，y0，z0)及其半径 max_radius，对中心点 **p0** 进行世界坐标到相机坐标变换：

$$\mathbf{p1} = \mathbf{p0} * \mathbf{T}_{wc}$$

p1 是物体的包围球球心的相机坐标，如图 6.27 所示。接下来定义 6 个点，它们与 **p1** 之间的连线分别平行于±x 轴、±y 轴和±z 轴，到 **p1** 的距离都为 max_radius；这些点定义了一个以 **p1** 为球心的包围球 S。这样，便可以通过一些简单测试来判断包围球是否在视景体内。

图 6.27 是一个 x-z 平面俯视图；对于 y-z 平面，测试方法与此相同，这里不再重复。需要考虑的情况有多种：对包围球和视景体进行比较，以判断它是完全在视景体外、完全在视景体内还是部分在视景体内。如果包围球 S 部分或完全落在视景体内，只需在数据流中包含整个球体的网格，然后执行其他测试。此时，可以判断包围球是否部分或全部位于视景体内，也可以判断包围球是否完全在视景体外，采用何种方式完全取决于您。然而，有时候使用负逻辑的开销更低，因此这里将采用这种方法。

在 2D 空间中，可以这样描述问题：给定一个圆，其圆心为 **p1**(x1，z1)，4 个点定义了其 x 和 z 方向的边界，即已知条件如下：

- 圆心：**p1**(x1, z1)；
- 圆周上的 4 个点：

—**p2**(*x1*, *z1*+max_radius);
—**p3**(*x1*, *z1*-max_radius);
—**p4**(*x1*+max_radius, *z1*);
—**p5**(*x1*-max_radius, *z1*)。

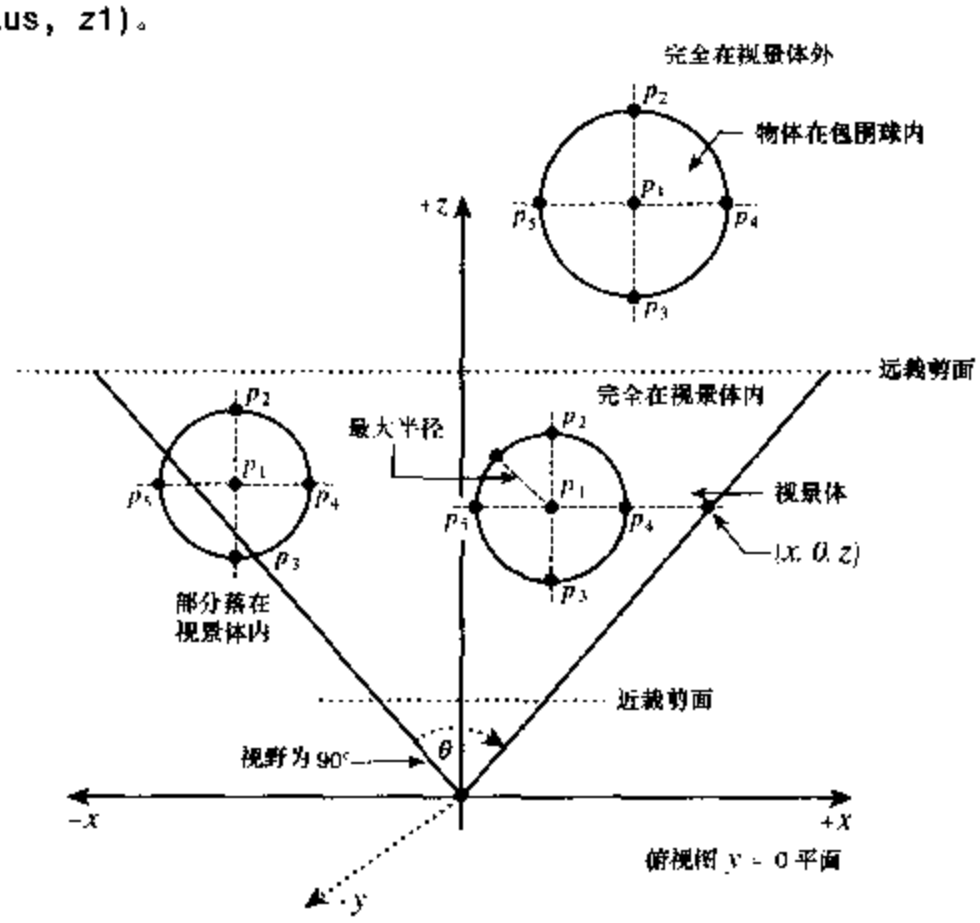


图 6.27　包围球测试细节

理论：如果这 5 个点都位于视景体内，则包围圆/球完全位于视景体内。

推论：如果这 5 个点都位于视景体外，则包围圆/球完全位于视景体外。

作者通常使用上述推论来排除整个球体。为此，需要根据视景体来得出一些方程，以便能够通过将点 **p1-p5** 代入方程，快速做出判断。根据图 6.26 可知，判断这些点与近裁剪面和远裁剪面的相对位置非常容易。如果下述任何一个条件为真，则可以肯定物体不在视景体内。

对于远裁剪面，条件为：

```
If (p3.z > far_z)
   {
   // 物体不在视景体内
   } // end if
```

对于近裁剪面，条件为：

```
If (p2.z < near_z)
   {
   // 物体不在视景体内
   } // end if
```

问题在于，视景体的左平面、右平面、上平面和下平面是一般性平面。然而，可以使用三角形（或平面方程）和点积来判断包围球和这些平面之间的相对位置。这里只考虑 2D 情形，且假设视野为 90 度。这样，问题简化为点 **p**(*x*, *z*) 是否在视景体左平面和右平面定义的三角形中。

视野为 90 度时，视景体左右平面的平面方程为

$$|x| = z$$

因此对于点 **p**(*x*, *z*)，如果 *x*<*z*，则该点位于右裁剪面的左边；如果 *x*>*z*，则位于右边；如果 *x*=*z*，则刚好位于裁剪面上。对于左裁剪面，判断方法与此相同，但需要对 *x* 求负。

因此，对于右裁剪面，判断条件如下：

```
if (p5.x > p5.z)
   {
```

```
// 物体不在视景体内
} // end if
```

对于左裁剪面，判断条件如下（这里对 z 进行了求负，因为左裁剪面的方程为 $-z=y$）。

```
if (-p4.x > p4.z)
   {
   // 物体不在视景体内
   } // end if
```

对于上裁剪面和下裁剪面，也可以采用相同的方法；因此判断点 $\mathbf{p}(x, y, z)$ 是否位于视景体外的通用规则如下：

```
if ((z > far_z)   || (z < near_z) || // 远、近裁剪面
    (fabs(x) < z) || // 左、右裁剪面
    (fabs(y) < z))   // 上、下裁剪面
    {
    // 物体不在视景体内
    } // end if
```

现在，只需根据前面的规则对点 **p1**～**p5** 进行测试即可。

注意：这种方法看似不错，但别忘了它依赖于数据结构和物体的表示方法。游戏空间可能是由无约束的多边形（free polygon）组成的，因此没有真正可包含的物体。也许删除背面是没有意义的，因为网格是基于顶点列表的，必须对其进行变换，因此没有独立于构成物体的顶点的多边形列表。或者所有的物体都是半透明的，因此所有多边形都有两面。总之，在很多情况下，背面消除是没有意义的。

3. 物体消除的局限性

关于物体消除，需要指出的一点是，如果物体的包围球不在视景体内，则物体也不可能在视景体内；反之则不正确：如果包围球部分位于视景体内，并不意味着物体也部分位于视景体内。这是因为包围球可能并没有很好地代表物体。例如，请看图 6.28，其中有一个非常长的物体的包围球以及一个视景体。有趣的是，包围球部分位于视景体内，但物体并不在视景体内。这并不意味着包围球测试不管用，而只是意味着它可能无法剔除所有位于视景体外的物体。因此，有时候将使用与物体形状更匹配的包围几何体，如立方体或平行六面体。

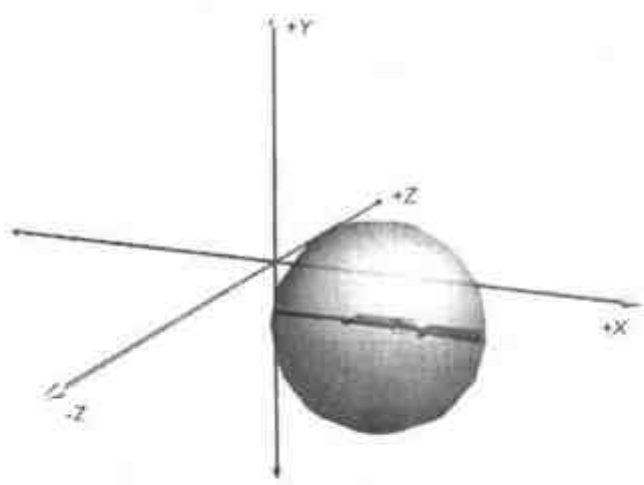


图 6.28　包围球可能不是最好的选择

至此，对物体和背面消除的简单介绍就结束了。图 6.29 是更新后的 3D 流水线。要将图像渲染到屏幕上，我们还有很长的路要走。

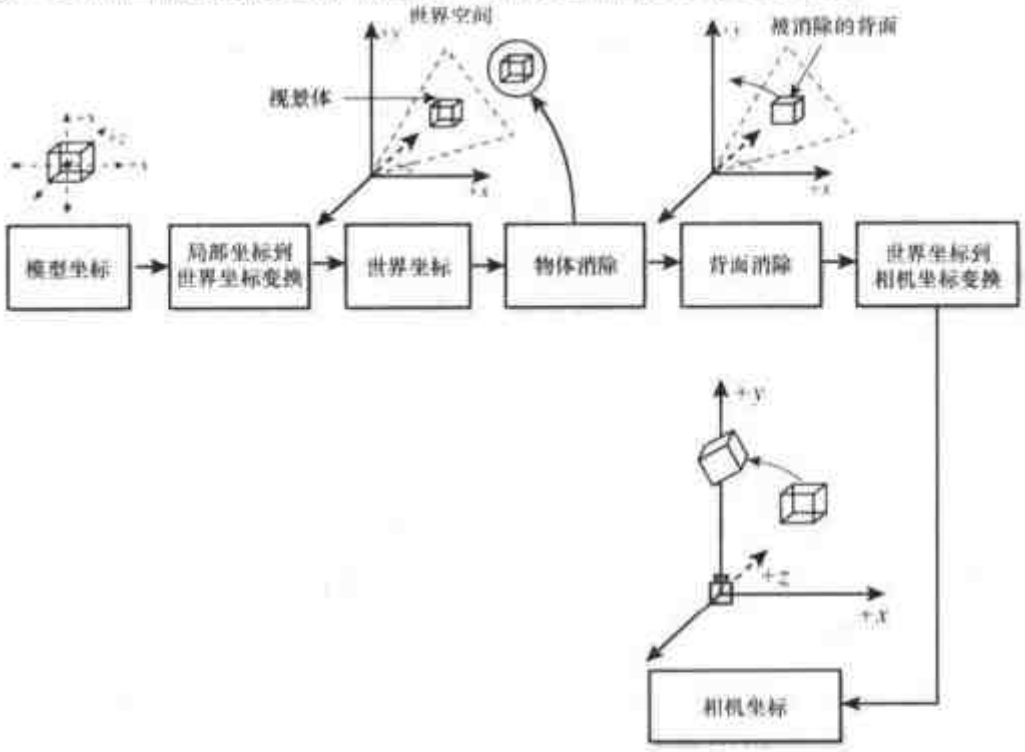


图 6.29　包含背面消除和物体消除的 3D 流水线

6.3.6 透视坐标

我们来回顾一下 3D 流水线的主要阶段。

- 第一阶段：在各个 3D 物体局部空间内使用局部坐标系定义它们。这种表示方法被称为局部/模型坐标表示法。
- 第二阶段：定义好所有 3D 模型后，通过将模型坐标变换为世界坐标，将它们放置到虚拟的 3D 世界中。这是通过模型坐标到世界坐标变换实现。这种变换只不过是根据要将物体中心放到世界空间的什么位置，相应地平移物体的顶点而已。在处理过程中，不要修改模型坐标，而应使用另一个数组来存储处理结果。
- 第三阶段：物体被放置到世界空间中后，它们是由世界坐标定义的，您需要通过一台虚拟相机来观察世界。表示相机的方式很多，但只要有位置和朝向就足够了。
- 第四阶段：此时可能需要执行隐藏物体/面消除，以最大限度地减少需要将其世界坐标变换为相机坐标的几何体数目。可能首先剔除物体，然后对于余下的可见物体，剔除其背面。
- 第五阶段：通过虚拟相机的镜头观察物体的方式很多，但当相机位于原点，镜头指向+z 轴方向时，可极大地简化处理工作。因此，对游戏世界中的几何体进行世界坐标到相机坐标变换，使得相机位于原点，镜头指向+z 轴。这种变换是用于放置虚拟相机时使用的平移变换和旋转变换的逆变换。
- 第六阶段：最后，相机位于原点，且观察角皆为 0 后，需要定义一个视景体，用于表示通过相机的虚拟镜头能够拍摄到的范围。视景体是由近裁剪面和远裁剪面以及水平和垂直视野定义的。位于视景体内的物体将被投影并渲染到屏幕上。投影阶段就是接下来要讨论的内容。

1. 准备工作

讨论透视坐标之前，先简要地回顾一下投影过程和视景体。假设要渲染一个由一组顶点（v[ ]）组成的 3D 物体，而所有的变换工作已经完成。这个物体位于如图 6.18 所示的视景体中。问题是，如何将该图形投影到屏幕上？

答案取决于您想看到什么。我们首先来回顾一下虚拟相机系统，以方便解释相机、世界和屏幕之间的关系。

假设这个视野为 90 度的相机系统的观察方向与 Z 轴平行，并指向+Z 轴方向。下面的讨论将基于这个相机系统。

2. 投影面/视平面

视景体决定了需要对哪些物体进行投影，而 3D 世界的虚拟图像将被投影到视平面上，然后被显示到 2D 光栅屏幕上。我们已经决定，相机最终将被放置到(0，0，0)处，并指向+Z 轴方向。接下来需要指定远裁剪面、近裁剪面和视平面。当然，您可以随意指定，但视平面的选择决定了观察流水线中后续处理的难易程度。

例如，由于视野为 90 度，因此可以通过 3D 系统的俯视图（如图 6.30 所示）来得出某些结论，同时又不失普遍性。然后将这些结论推广到 3D 空间中。首先，视平面和视点(0，0，0)之间的距离有时被称为视距(d)。d 的值对投影变换及其与屏幕坐标之间的关系有重大影响。

定义好上述值并将 3D 物体放到视景体内后，为将其渲染到视平面上，我们从物体上的每点发出一条精确的投影线（mathematical projector），它穿过视平面，最终到达视点。这就是投影。然后，在虚拟视平面上形成的图像被渲染到计算机屏幕上。

当然，这里省略了一些与缩放相关的细节，稍后将介绍这些细节。接下来，更详细地讨论透视变换。

3. 透视变换

透视变换指的是将物体的顶点投影到视平面上，如图 6.31 所示。具体描述如下：给定视平面与投影点（相机视点）之间的距离 d，可以很容易计算出物体上任何一点和视点之间的连线与视平面的交点。

对于这个几何问题，可以在 2D 空间中进行解决，然后推广到 3D 空间。图 6.31 是前述 3D 系统的侧视图（观察方向垂直于 *y-z* 平面）。我们要将点 **p**(*y*0，*z*0)投影到视平面上，后者离原点的距离为 d，其平面方程为 *z* = d。数学公式很简单，只是两个相似三角形，即：

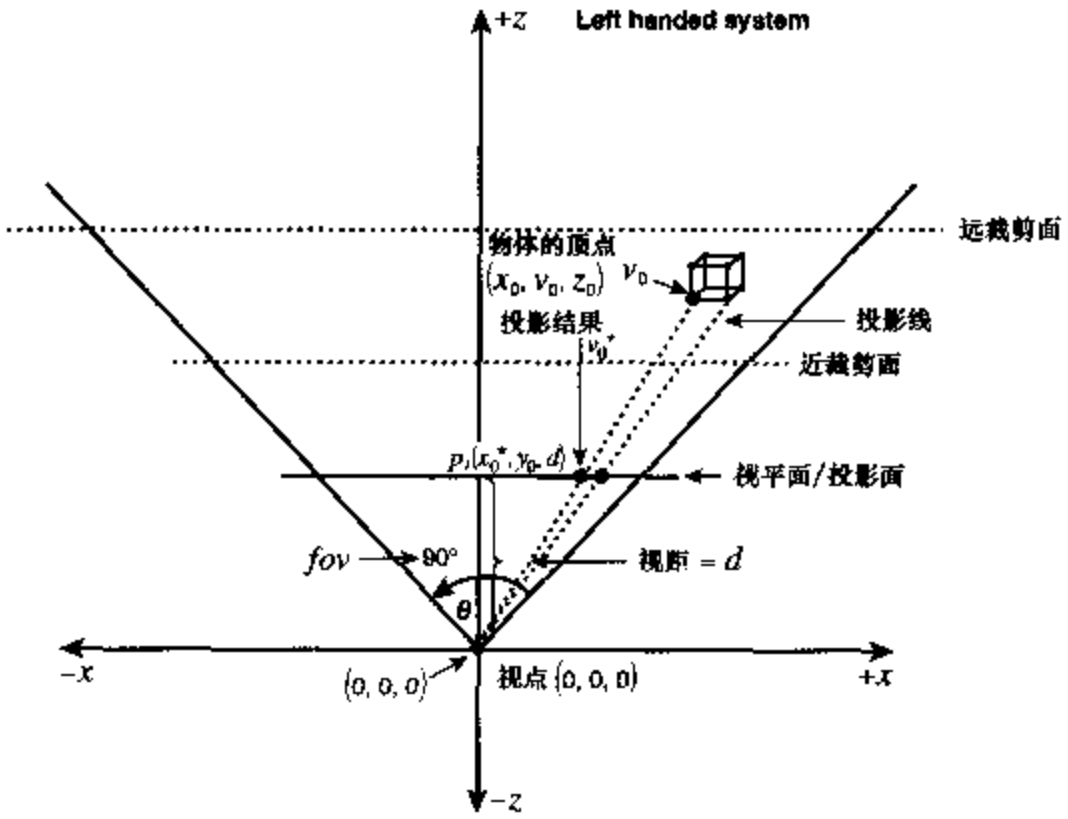
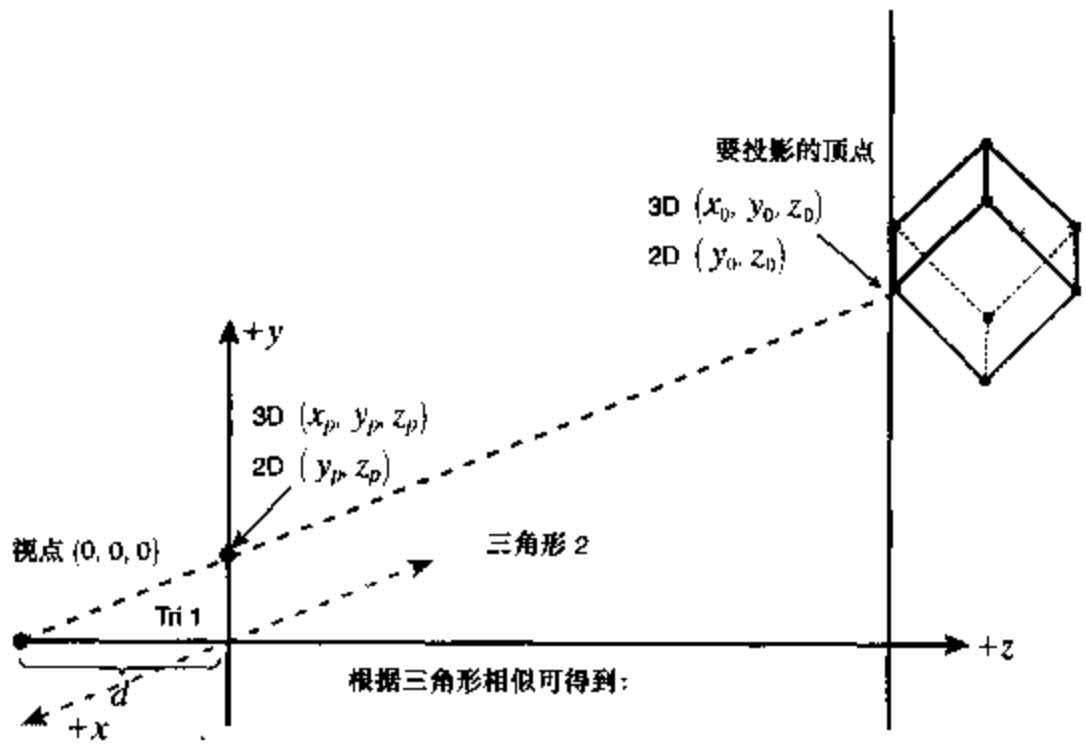


图 6.30　简化的 x-z 平面上 2D 视景体投影



$$xp = \frac{d \bullet x_0}{z_0} \qquad xp = \frac{d \bullet y_0}{z_0}$$

图 6.31　透视变换

$d/z0 = y_p/y0$

对其进行整理可得到：

$y_p = d*y0/z0$

同样，在 *x-z* 平面上，存在如下投影关系：

$d/z0 = x_p/x0$

对其进行整理可得到：

$x_p = d*x0/z0$

将上述公式组合在一起，可得到视平面为 z=d，视点位于 (0，0，0) 处时的投影变换：

$x_{per} = d*x/z$
$y_{per} = d*y/z$

首先说明一下上述公式存在的问题。当 z=0 时，上述公式的结果为无穷大，这很糟糕。也就是说，不能对 z=0 的顶点进行投影。另外，z 值为负的物体将被倒转，但能够被投影。这就是需要指定近裁剪面的原因，通过指定近裁剪面，可避免这些情况发生。

注意：还有一种平行（正交投影），这种投影有时被用于机械和 CAD 程序中。这种投影只是丢弃 z 坐标。当然，没有使用透视，但设计零件和住宅时，常常不希望进行透视。因此在投影阶段，顶点（$x0$，$y0$、$z0$）被投影到（$x0$，$y0$），有时候 $z0$ 被用作亮度值，根据 z 距离的远近给图像着色。

下面从编码和矩阵的角度来探讨投影变换。从编码的角度说，投影变换非常简单；但必须使用齐次坐标。原因在于，使用一个不包含 z 值的矩阵时，难以得到除以 z 的结果。下面是使用蛮干方式进行投影变换的代码：

```
POINT3D cube_camera[8]; // 用于存储相机坐标
POINT3D cube_per[8];    // 用于存储透视坐标

// 对相机坐标执行透视变换
// 假设视距为 d
for (int vertex = 0; vertex<8; vertex++)
    {
    float z = cube_model[vertex].z;

    cube_per[vertex].x=d*cube_camera[vertex].x/z;
    cube_per[vertex].y=d*cube_camera[vertex].y/z;

    // 不需要 z 坐标，因此复制原来的值即可
    cube_per[vertex].z=cube_camera[vertex].z;
    } // end for
```

至此，我们完成了投影变换，但如果要创建一个完全基于矩阵的系统，必须使用矩阵表示所有操作。下面的矩阵 T_{per} 用于对 4D 齐次坐标执行投影变换：

$$T_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

下面使用相机坐标为 $(x_c, y_c, z_c, 1)$ 的点 p 来检验一下它是否管用：

$$p * T_{per} =$$

$$[x_c \ y_c \ z_c \ 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= [x_c \ y_c \ z_c \ (z_c/d)]$$

如果将分量 x、y 和 z 分别除以分量 $w(z_c/d)$，将齐次坐标转换为常规坐标，结果将如下：

$$=[x_c*d/z_c,\ y_c*d/z_c,\ z_c*d/z_c]$$

当然，我们无需考虑坐标 z，因为投影后，我们只关心坐标 x 和 y，它们的值如下：

$$x = x_c*d/z_c$$
$$y = y_c*d/z_c$$

这就是使用矩阵来执行投影变换。这是我们第一次将 4D 齐次坐标转换为 3D 坐标。我们可能重新审视考虑是否需要使用矩阵来执行投影变换，答案取决于将如何创建引擎。也就是说，要创建一个完全基于矩阵的系统，还是对于大部分变换使用矩阵来执行，但为提高速度，使用简单的代码来完成诸如投影变换和屏幕变换等操作？现在我们不考虑这种问题，但它确实是个问题。

我们几乎就要降服投影这条巨龙了，但还有一个细节需要考虑，这就是视距 d。应使用什么样的 d 值呢？这个问题问得好，但读者可能不喜欢答案：任何值都可以。是的，可以根据自己的偏好选择任何 d 值。

改变 d 值将改变图像的比例。那么，什么样的 d 值最合适呢？作者通常采用两种方式之一来选择 d 值。首先，如果想简化数学计算，将 d 值设置为 1，这样投影公式将变成：

$$x_{per} = 1*x/z$$
$$y_{per} = 1*y/z$$

而投影变换矩阵将变为：

$$T_{per1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

执行上述变换时，得到的视平面坐标将是归一化的。换句话说，任何顶点被投影后，坐标 x 和 y 都在 -1 到+1 之间。另外，需要将视平面坐标映射到屏幕坐标，以便查看。也就是说，将 x 和 y 坐标范围-1 到+1 映射到屏幕的分辨率宽度×高度。这里存在一个问题，因为我们要将一个方形视平面映射到视口，因此后者也必须是方形的，否则某个轴将被变形。因此，如果视口或屏幕不是方形的，必须将视平面的宽高比设置成与视口相同，或者在执行透视坐标到屏幕坐标变换期间，对某个轴进行缩放。有关这一点，将在后面更详细地介绍；但读者牢记。

另一种设置 d 值的方法是，根据屏幕坐标范围来计算。如果屏幕的分辨率为 640×480，可以这样选择 d 值，即使投影坐标位于范围 640×480 内。然而，这里存在一个小问题，因为 x 和 y 方向的范围不同。这意味着，对于 x 轴投影变换和 y 轴投影变换，需要使用不同的 d 值，但如果这样做，将导致最后渲染到屏幕上的图像变形。因此，投影到某种分辨率的屏幕上，并计算确保 x 轴的 FOV 为特定值的 d 值时，必须使用相同 d 值来进行 x 投影变换和 y 投影变换，这样 y 轴的视野将与 x 轴的视野不同。

当投影视平面或视口不是方形的时，方形投影不再适合，有一个将 x 轴和 y 轴关联起来的因子，它就是宽高比。对于 640×480 的屏幕，宽高比为 4:3（1.33333）。如果视平面不是方形，执行透视坐标到屏幕坐标变换时，宽高比将派上用场。在这种情况下，执行透视计算时，必须将视距和宽高比相乘。

图 6.32 说明了如何计算 d 值。对于垂直轴，使用相同的 d 值，因为必须使用相同的视距。

注意：一种副作用是，垂直视野通常与水平视野不同，因为宽高比不为 1。

$$d = 0.5*\text{宽度}*\tan(\theta_h/2)$$

也就是说，视距等于半视角的正切与投影面宽度的乘积的一半。如果读者理解不了，请看图 6.32。$\tan(\theta)$ = 对边/邻边 $= y/x$；在这里，y 为视距，x 与宽度相关。

为推广变换公式，需要考虑宽高比，即将 y 分量乘以宽高比 ar。这样，变换公式如下：

$$x_{per} = x*d/z$$
$$y_{per} = y*ar*d/z$$

相应的投影变换矩阵如下：

$$T_{perd} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d*ar & & \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

注意：这里重新调整了 d 项的位置。

给定点 p[x_c y_c z_c 1]，将其乘以上述矩阵，看看结果是否正确：

$$p * T_{perd}$$
$$= [x_c \ y_c \ z_c \ 1] * \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d*ar & & \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$
$$= [(d*x_c)(d*ar*y_c)z_c \ z_c]$$

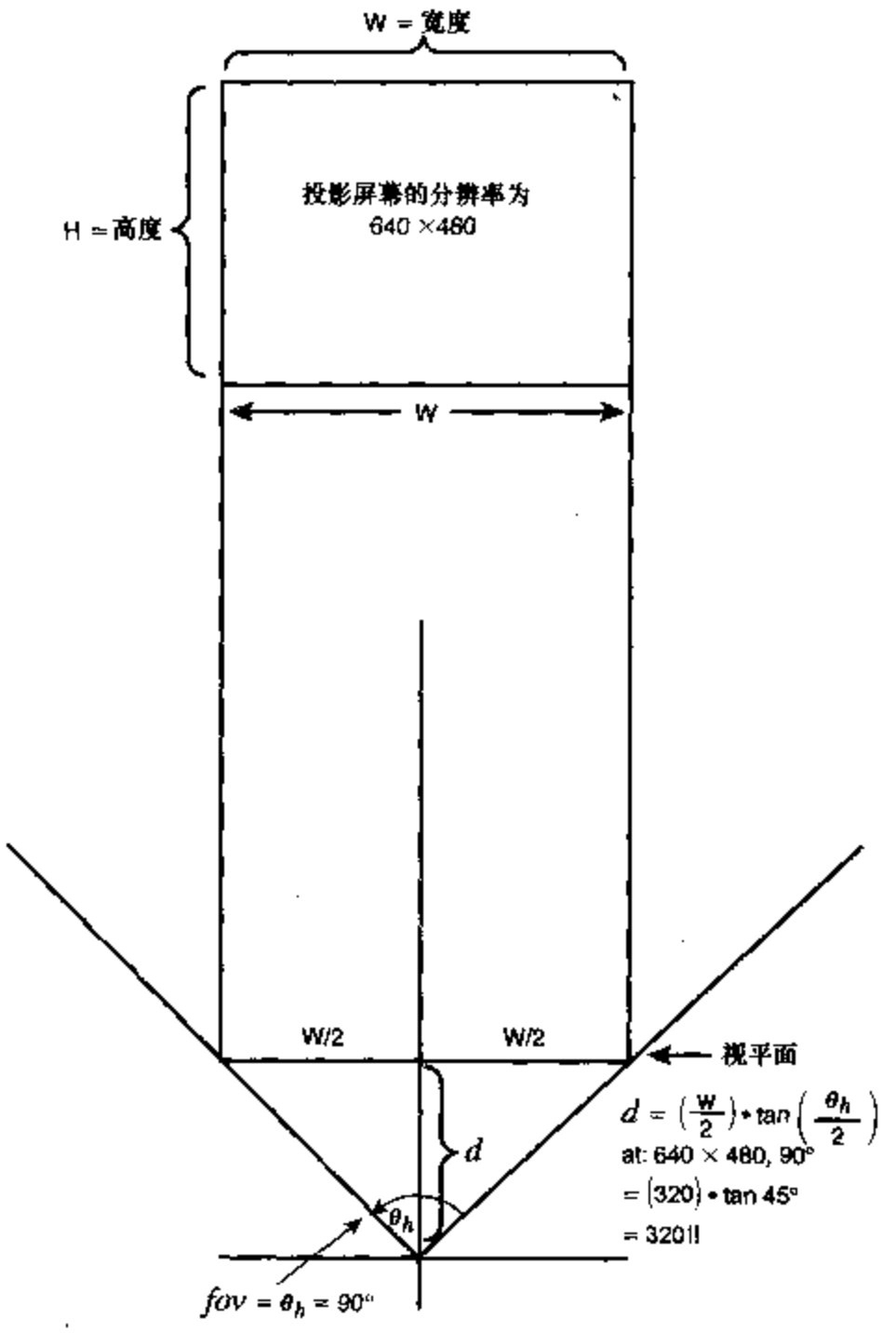


图 6.32　根据屏幕分辨率和视野计算 d 值

将分量 x 和 y 分别除以分量 $w(z_c)$，结果如下

$x = d*x_c/z_c$
$y = d*ar*y_c/z_c$

与前面相同，这表明该投影变换矩阵是正确的。

关键之处在于，可以手工计算投影变换，也可以使用矩阵来计算。读者可能会问，为何要使用矩阵呢？原因在于矩阵更方便，可以将多个矩阵组合起来。因此，可以将物体局部变换、局部坐标到世界坐标变换、世界坐标变换到相机坐标变换以及相机坐标到投影坐标变换组合成一个矩阵。

4. 透视投影和 3D 裁剪

简单地说，透视投影是将视景体内用相机坐标表示的点变换到视平面上，以便下一步将其变换为屏幕坐标。然而，透视投影还有另一种裁剪方面的用途：将棱锥形视景体变成长方体。

在图 6.33 中，有两个点：**p1**($x1$，$y1$，$z1$)和 **p2**($x2$，$y2$，$z2$)，它们定义了一条线段，这条线段的一部分位于视景体内。假设我们有一个线框 3D 引擎，因此需要绘制该线段。使用物体消除无法剔除该线段，因为它并没有定义一个物体；同时直线没有背面，因为它不是多边形。

绘制该直线时，需要使用视景体对其进行裁剪。这一步很重要，因为所有完全位于视景体内的物体都将被投影和渲染；同样所有完全位于视景体外的物体都将被剔除。然而，对于部分落在视景体内的几何体，必须对其进行裁剪，就像在 2D 窗口中裁剪直线和位图一样。当然，这里进行了简化，在实际工作中，必须使用视景体对多边形进行裁剪。

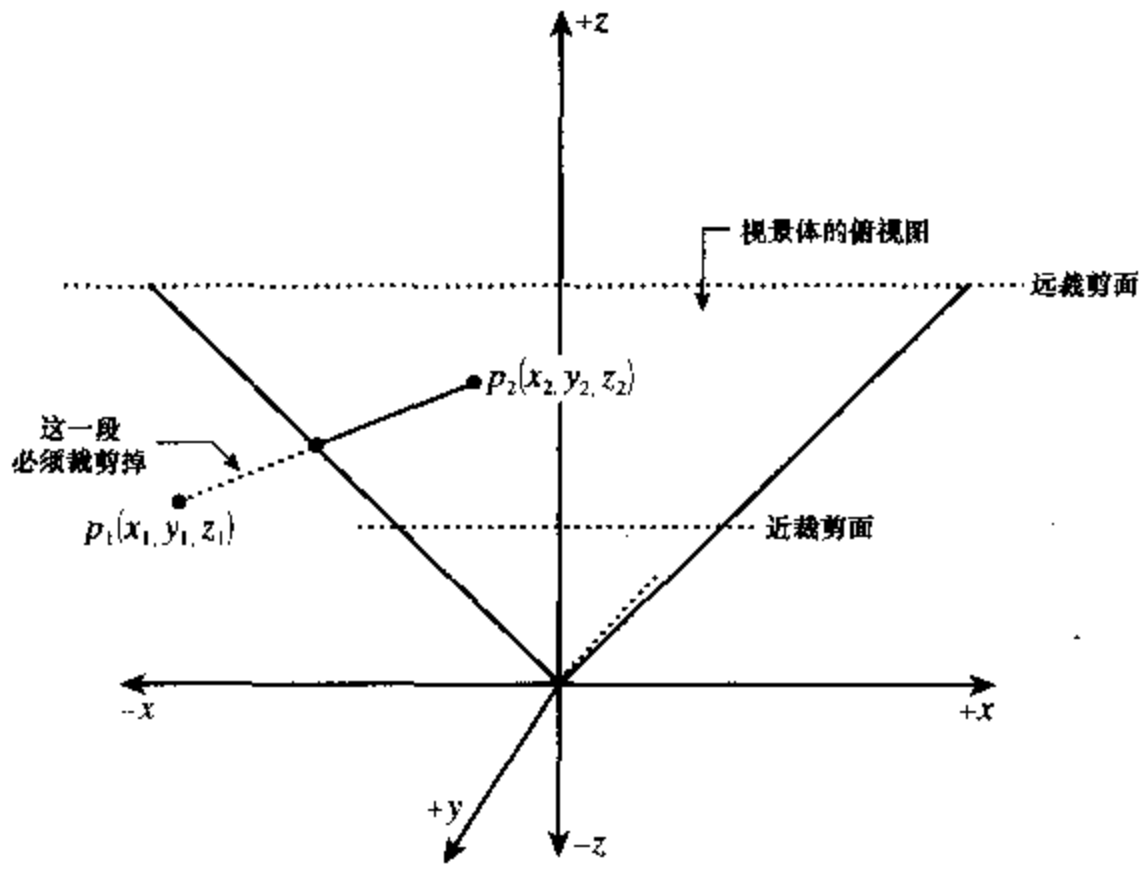
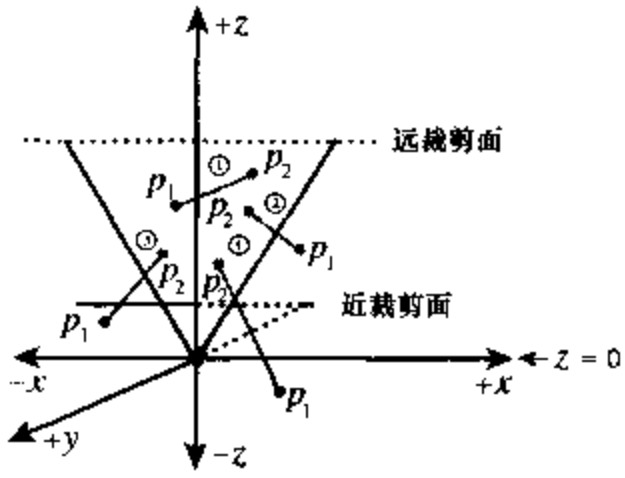
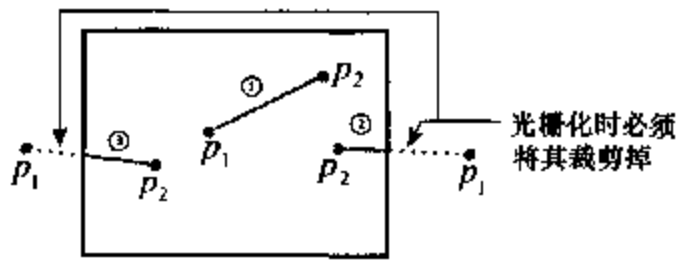


图 6.33　简单裁剪范例

对于直线裁剪，处理方式有两种。第一种方式是，不管三七二十一，对直线执行投影变换（假设两个端点的 z 坐标都大于零），到 3D 流水线的光栅化阶段，再在 2D 空间内使用窗口对直线进行裁剪。这被称为图像空间裁剪。换句话说，图像空间裁剪是在所有要渲染的物体都被转换为屏幕坐标后（虽然它们可能在屏幕外），再使用屏幕空间或视口对它们进行裁剪，如图 6.34 所示。



第 1 步：线段 1、2 和 3 的 z 坐标都大于零，但线段 4 并非如此，因此投影线段 4 将带来问题



第 2 步：以透视方式将线段 1、2 和 3 投影到屏幕空间中，并在光栅化期间进行裁剪

图 6.34　图像空间裁剪流水线

图像空间裁剪很管用，也易于实现，但这意味着需要在 2D 空间内对每个物体进行裁剪，即使它完全位于窗口内。这将极大地增加处理器的负载，尤其是渲染多边形时，需要对每个像素进行测试。但绘制直线时，在 2D 空间内进行图像空间裁剪的开销可能比在 3D 空间内进行裁剪时低，稍后将解释其原因。

第二种裁剪方式是在物体空间（即物体或几何体所在的数学空间）中进行。消除物体和/或背面，以免将其变换为相机坐标的操作就是一种物体空间操作。然而有些物体和几何体在游戏中是可见的，否则不称其为游戏。在这些物体中，有些并不完全落在视景体内，因此必须使用视景体对它们进行裁剪。这样，问题被简化为：如果在对几何体执行投影变换之前，要执行物体空间裁剪，必须在 3D 空间内使用视景体对所有不完全位于视景体内的几何体（直线或多边形）进行裁剪。这被称为 3D 裁剪。除非您是 Chris Knight，否则这项工作可不容易。

我们使用的视景体非常简单：有远裁剪面和近裁剪面，水平和垂直视野都是 90 度（上下平面和左右平面的平面方程分别是 $|x|=z$ 和 $|y|=z$），并假定投影面是方形的；然后使用这 6 个面对 3D 线段进行裁剪时仍非常繁琐。如果视平面不是方形的，则上下平面和左右平面的平面方程分别是 $|x|=z$ 和 $|y|=z/ar$。

另外，如果水平视野和垂直视野不是 90 度，裁剪操作将更复杂，因为上下平面和左右平面的平面方程将为通用形式。

如果首先在 3D 空间内使用视景体对所有部分位于视景体内的几何体进行裁剪（物体空间裁剪，如图 6.33 所示），可保证所有被投影后的几何体都将位于视平面内。正如前面指出的，3D 裁剪比较棘手，在本书后面介绍这个主题时，读者将发现，不但需要裁剪多边形顶点，还需要裁剪纹理坐标和光照坐标。

然而，在 3D 空间中，使用立方体和长方体进行裁剪要简单得多，因为这相当于在每个平面内使用矩形进行裁剪。在作者看来，投影变换和裁剪之间的关系是，如果愿意对所有可能看得见的东西都执行投影变换，投影变换本身就会将视景体变成一个长方体，原来呈一定角度（在我们的例子中为 90 度）的四个面将两两平行，这样将得到一个立方体，可以使用它来裁剪投影坐标。很惊讶吧！我们来看一个例子，以检查这种说法是否正确。

由于每个面的情况都相同，我们以视景体的俯视图（*x*-*z* 平面）为例，如图 6.35a 所示，其中远裁剪面为 $z = \text{far\_}z$，近裁剪面为 $z = \text{near\_}z$。投影面/视平面为 $d = 1$。当然，视野为 90 度，我们假设投影面为方形。我们要做的是，对视景体（2D）的角点执行透视投影，看看变换后是什么样子。也就是说，对其进行透视变换后，视景体本身是什么样子？

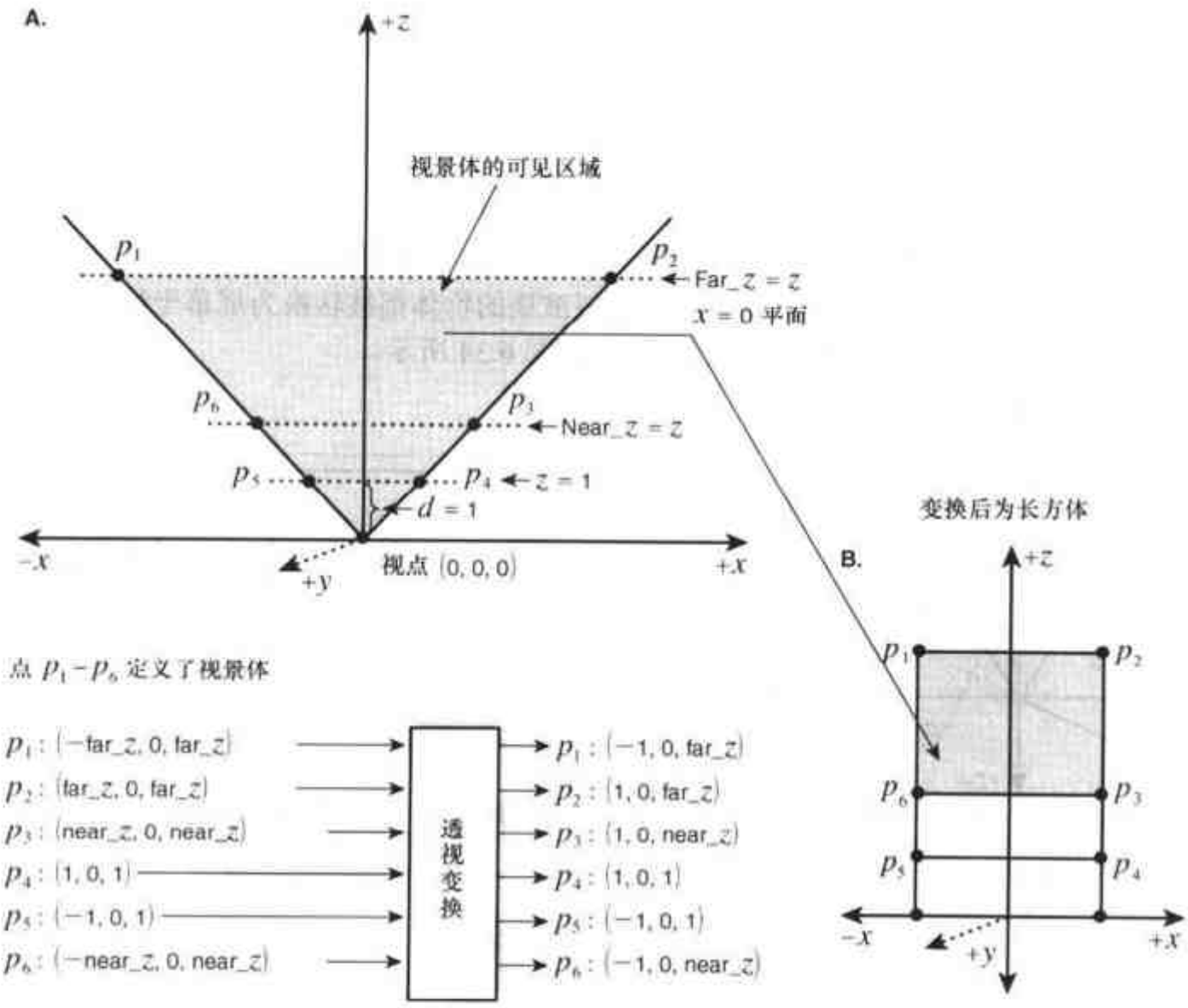


图 6.35　透视变换将视景体变成长方体

点 **p1**～**p6** 定义了视景体在 x-z 平面上的 2D 投影图，它们的 y 坐标都是 0：

```
p1(-far_z, 0, far_z)
p2(far_z, 0, far_z)
p3(near_z, 0, near_z)
p4(1·0,1)
p5(-1,0,1)
p6(-near_z, 0, near_z)
```

提示：读者可能注意到了，有些点的 x 坐标使用 z 坐标表示的，这是因为视野为 90 度，投影平面为方形，因此视景体左右平面和上下平面的平面方程分别为 $|x| = z$ 和 $|y| = z$，所以 $\{x, y\} = z$。

如果将 d 值设置为 1，并对这些点进行投影变换，即将每个分量乘以 d 再除以 z 坐标，将得到如下结果：

```
p1'(-far_z/far_z, 0/far_z, far_z)          = (-1, 0, far_z)
p2' (far_z/far_z, 0/far_z, far_z)          = (1, 0, far_z)
p3' (near_z/near_z, 0/near_z, near_z)      = (1, 0, near_z)
p4' (1/1, 0/1, 1)                          = (1, 0, 1)
p5' (-1/1, 0/1, 1)                         = (-1, 0, 1)
p6' (-near_z/near_z, 0/near_z, near_z)     = (-1, 0, near_z)
```

当然，可以忽略分量 y，因为知道这是在 x-z 平面上。

现在绘制出点 **p1'～p6'**，结果如图 6.35b 所示，它们组成了一个矩形。经过变换，棱锥变成了矩形，使用后者来裁剪将很容易，即使是在 3D 空间中。当然，可以在 *y-z* 平面进行相同的分析，得到相同的结果。将这些结果合并起来，将得到一个长方体形视景体，如图 6.35b 所示。这里旨在向读者证明，可以使用透视变换来归一化所有的几何体，从而很容易地对它们进行裁剪。本书后面将介绍如何这样做。

总之，3D 流水线中不仅包含透视变换，还可能包含 3D 裁剪阶段（如果不想在图像空间中进行裁剪的话），此时的 3D 流水线如图 6.36 所示。另外，如果愿意对所有几何体都进行透视变换，可以在透视变换之后再进行 3D 裁剪，图 6.36 也说明了这一步。

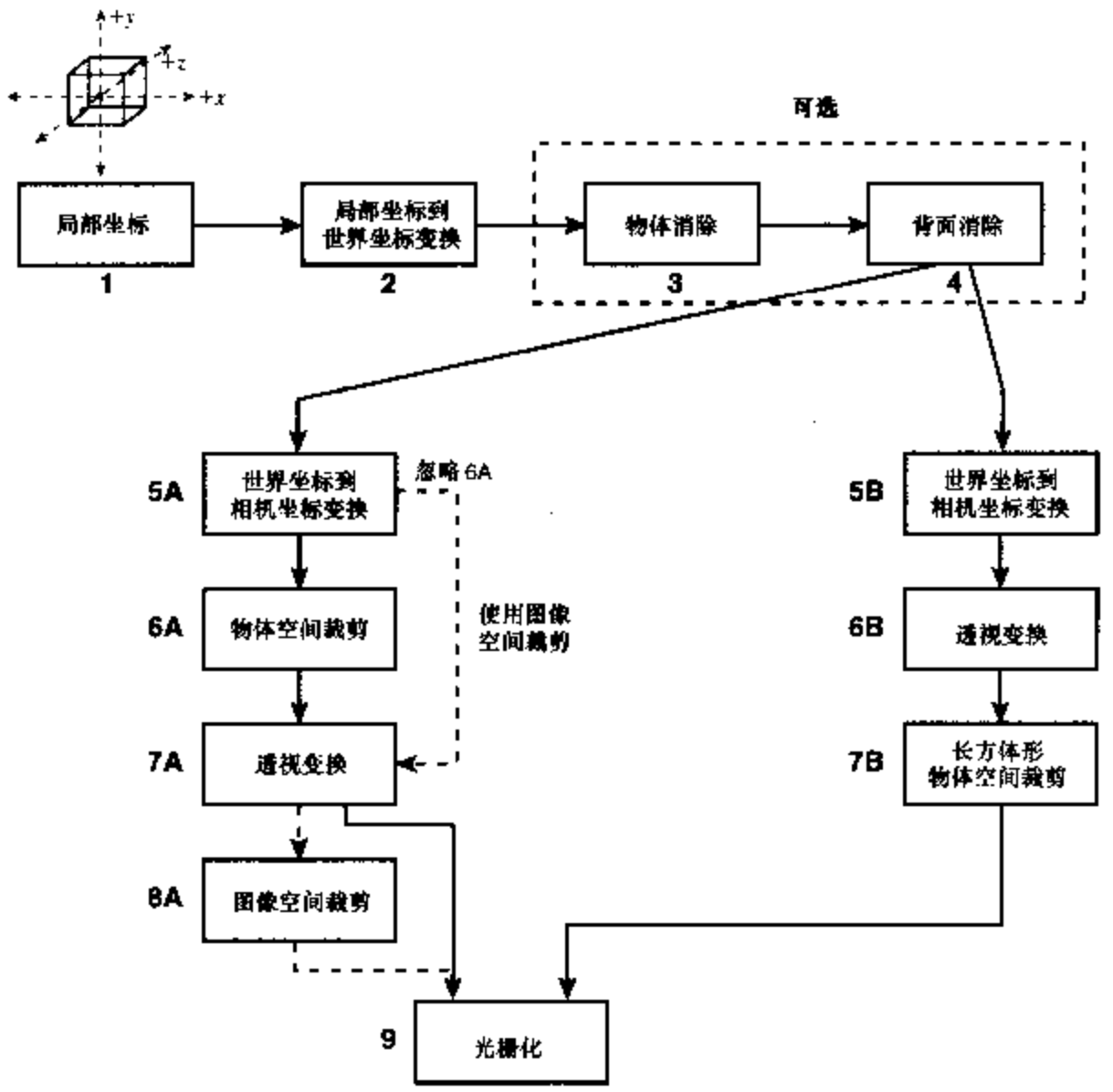


图 6.36　包含透视变换和裁剪的 3D 流水线

接下来介绍观察流水线的终点——屏幕坐标。

6.3.7　流水线终点：屏幕坐标

介绍流水线的屏幕坐标变换阶段之前，需要指出的是，我们省略了 3D 流水线中的光照和纹理映射阶段。这些高级主题不适合在本章介绍，但需要指出的是，在得到世界坐标后和得到屏幕坐标之前，必须对多边形进行光照计算和纹理映射。但现在暂时不去管它们，因为这里重点要介绍的是坐标变换。

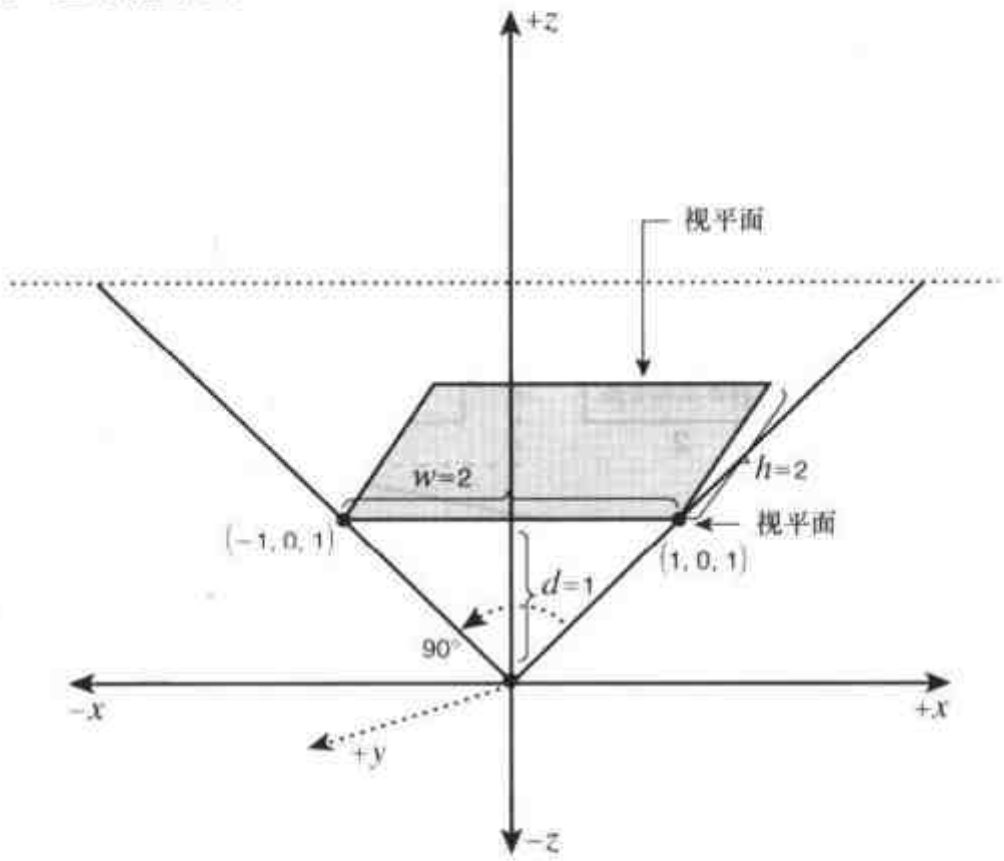
现在，我们得到了一系列的点，它们表示了我们想在屏幕上看到的东西。这些点经过了投影变换，可能已经根据屏幕大小对它们裁剪过，也可能没有裁剪过。现在假设没有根据屏幕裁剪过，并需要根据屏幕进行裁剪。屏幕坐标的计算依赖于执行的投影变换类型，即在虚拟空间中，视平面为多大？这是因为需要

将视平面映射到屏幕。下面来看几种透视变换。

1. 视野为 90 度且视距为 1

当视野为 90 度，视距为 1，且投影面为方形时，投影变换将所有几何体投影到一个 2×2 的虚拟视平面上（x 和 y 坐标的范围都是−1 到+1）。当投影面不是方形时（这种情况更常见），投影变换将所有东西都投影到一个 2×（2/ar）的虚拟视平面上（x 坐标的范围为−1 到+1，y 坐标的范围为−1/ar 到+1/ar），其中 ar 为宽高比。进行屏幕变换时，我们并不关心宽高比，因为我们假定在投影变换中已经考虑过宽高比。然而，必须在某个阶段考虑宽高比，否则图像将变形。

为让读者对投影有清晰的认识，图 6.37 对其做了说明。接下来需要将视平面坐标（方形投影时为 2×2，非方形投影时为 2×（2/ar））映射为视口（或屏幕）坐标。视平面位于虚拟相机空间内，其大小可能与视口（光栅化窗口）相同，也可能不同。



视野＝90°且宽度比＝1.0 时
视距为 1 的视平面为 2×2

视野为 90° 且宽度比不为 1 时
视距为 1 的视平面为 2×2/0.5

图 6.37　视野为 90 度且视距为 1 时投影到 2×2 投影面上的细节

对这种变换进行解释的方式有多种，我们假设需要渲染到一个视口缓存中，其原点(0，0)位于左上角，大小为 SCREEN_WIDTH×SCREEN_HEIGHT，如图 6.38 所示。这里宽度和高度相等，与第 2 章和第 3 章使用的虚拟计算机系统设置相同。

因此，只需计算出这样的变换方式，即按下述方式将视平面坐标变换为屏幕（视口）坐标：

x_{per}: -1 到 1 -> x_{screen}: 0 到 SCREEN_WIDTH − 1
y_{per}: -1 到 1 -> y_{screen}: 0 到 SCREEN_WIDTH − 1

注意：y 轴被反转了。

推导数学公式的方式有多种，但最简单的方法是，找到屏幕中心，并从这里开始变换。然而，需要注意的是，y 轴被反转了，需要考虑到这一点。这样，变换公式如下：

$$x_{screen} = (x_{per}+1) * (0.5*SCREEN\_WIDTH - 0.5)$$
$$y_{screen} = (SCREEN\_HEIGHT - 1) - (y_{per} + 1) * (0.5*SCREEN\_HEIGHT - 0.5)$$

下面使用边界条件来检查上述公式是否正确，例如，将视平面的 4 个顶点变换为屏幕坐标：

```
p1(-1,1)   = p1'(0, 0)
p2(1,1)    = p2'(SCREEN_WIDTH − 1, 0)
```

```
p3(1,-1)  = p3'(SCREEN_WIDTH - 1, SCREEN_HEIGHT - 1)
p4(-1,-1) = p4'(0, SCREEN_HEIGHT - 1)
```

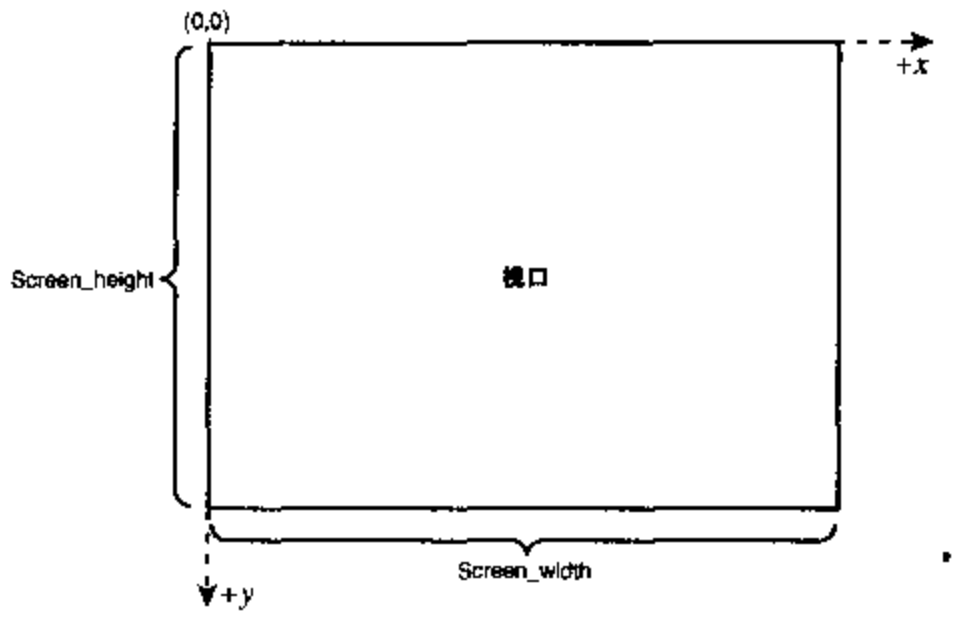


图 6.38　视口的定义

结果为屏幕的 4 个角。当然，可以采用不同方式对屏幕变换公式中各项相乘再相加，例如：

$$x_{screen} = (x_{per}+1) * (0.5*SCREEN\_WIDTH-0.5)$$
$$= (x_{per}) * (0.5*SCREEN\_WIDTH-0.5) + (0.5*SCREEN\_WIDTH-0.5)$$
$$y_{screen} = (SCREEN\_HEIGHT-1) - (y_{per}+1)*(0.5*SCREEN\_HEIGHT-0.5)$$
$$= -(y_{per}) * (0.5*SCREEN\_HEIGHT-0.5) - (0.5*SCREEN\_HEIGHT-0.5) + (SCREEN\_HEIGHT-1)$$
$$= -(y_{per}) * (0.5*SCREEN\_HEIGHT-0.5) + (0.5*SCREEN\_HEIGHT-0.5)$$

假定$\alpha = 0.5*SCREEN_WIDTH-0.5$，$\beta = 0.5*SCREEN\_HEIGHT-0.5$，则：

$$x_{screen} = \alpha + x_{per}*\alpha$$
$$y_{screen} = \beta - y_{per}*\beta$$

有趣的是，(α, β)正好是屏幕中心，这没什么好奇怪的，因为此时映射的是视平面点(0, 0)。上述公式的好处在于，将其转换为矩阵时将容易得多。由于处理的是 2D 屏幕坐标，因此没有必要使用 4D 齐次坐标。也就是说，如果透视点坐标格式为(x, y)，则没有必要使用 4×4 矩阵。如果愿意，可以使用 3×3 矩阵。有时候，全部使用 3×3 矩阵（丢弃 z 分量）或 4×4 矩阵是个好主意，具体采用哪种方式完全取决于您，但处于完整性考虑，这里将介绍这两种矩阵。

首先来看看 4×4 变换矩阵，假设视平面上点的透视坐标为$[x_{per}, y_{per}z, 1]$，其中 z 值无关紧要，可以为 0，因为其作用已经发挥过。由于这是视距为 1 时的透视坐标到屏幕坐标变换，因此将矩阵命名为 T_{scr1}：

$$T_{scr1} = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & -\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \alpha & \beta & 0 & 1 \end{bmatrix}$$

使用点 $p = [x_{per}\ y_{per}\ 0\ 1]$对上述矩阵进行检测：

$$p * T_{scr1}$$
$$= [x_{per}\ \ y_{per}\ \ 0\ \ 1] * \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & -\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \alpha & \beta & 0 & 1 \end{bmatrix}$$
$$= [x_{per}*\alpha+\alpha \ \ -y_{per}*\beta+\beta,\ 0,\ 1]$$

从中提取分量 x 和 y，结果为：

$$x = x_{per}*\alpha+\alpha$$
$$y = -y_{per}*\beta+\beta$$

该变换矩阵是正确的。当然，这有点多此一举，因为我们完全可以使用前面的公式，将变换矩阵抛到脑后。但是别忘了，通过组合，可以在一个矩阵中完成 50 个不同的操作，这就是我们一直在介绍如何使用矩阵来完成每项工作的原因。接下来看第二个例子，这里假设有一个 2D 点(x_{per}, y_{per})，需要使用矩阵将其

变换为屏幕坐标。问题在于，我们需要平移(α，β)，因此需要使用齐次坐标。我们将使用 2D 齐次坐标——它看起来像 3D 坐标，但坐标为(*x*，*y*，*w*)，然后使用一个 3×3 矩阵来执行屏幕变换。假设视距为 1，需要将点 $p(x_{per}, y_{per}, 1)$变换为屏幕坐标。变换矩阵如下：

$$\mathbf{T}_{scr1} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & -\beta & 0 \\ \alpha & \beta & 1 \end{bmatrix}$$

使用点 $\mathbf{p}=[x_{per}\ y_{per}\ 1]$对上述矩阵进行检测：

$$\mathbf{p} * \mathbf{T}_{scr1}$$

$$= [x_{per}\ \ y_{per}\ \ 1] * \begin{bmatrix} \alpha & 0 & 0 \\ 0 & -\beta & 0 \\ \alpha & \beta & 1 \end{bmatrix}$$

$$= [x_{per}*\alpha+\alpha,\ \ -y_{per}*\beta+\beta,\ \ 1]$$

从中提取分量 x 和 y，结果为：

$$x = x_{per}*\alpha+\alpha$$
$$y = -y_{per}*\beta+\beta$$

这表明该变换矩阵是正确的。最后，来看看执行这种变换的代码，这里假设透视坐标已经被存储在数组 cube_per[8]中：

```
POINT3D cube_per[8]; // 用于存储透视坐标

POINT2D cube_screen[8]; 用于存储屏幕坐标
// 对透视坐标执行屏幕变换
// 假设屏幕的大小为 SCREEN_WIDTH x SCREEN_HEIGHT
// d=1，因此透视坐标是归一化的，即取值范围为-1 到 1

// 预先计算 alpha 和 beta
alpha = (0.5*SCREEN_WIDTH-0.5);
beta  = (0.5*SCREEN_HEIGHT-0.5);
// 循环并变换
for (int vertex = 0; vertex<8; vertex++)
    {
    cube_screen[vertex].x=alpha+alpha*cube_per[vertex].x;
    cube_screen[vertex].y=beta-beta*cube_per[vertex].y;
    } // end for
```

正如读者看到的，手工执行这种变换（不使用矩阵）并不复杂。

注意：这里没有对视平面、视口和屏幕坐标进行定义。视平面是在虚拟空间中将物体投影到其上的平面；视口是视平面坐标被映射到其中的窗口（在很多情况下，就是屏幕窗口）。

2. 视野和视距为任何值

前面介绍了视距为 1，视野为 90 度，在透视空间中视平面坐标在每个轴上被归一化为−1 到 1（方形投影）时的透视坐标到屏幕坐标变换；下面讨论更普遍的情况，在这种情况下，我们这样来计算 d 值：

```
d = (0.5)*viewplane_width*tan(θh/2)
```

这里我们根据主轴（通常是 *x* 轴）的视野 θ_h（视平面为方形时为 90 度，但在这里可以为任何值）来计算 *d* 值。另外，还需要知道将最终图像投影到其上的视平面的大小，如果与屏幕（视口）相同，则为 SCREEN_WIDTH×SCREEN_HEITHG。然而，需要注意的是，屏幕坐标范围如下：

- x 坐标——0 到 SCREEN_WIDTH − 1；
- y 坐标——0 到 SCREEN_HEIGHT − 1。

因此，必须将前述公式中的 viewplane_width 和 viewplane_height 替换为 SCREEN_WIDTH − 1 和 SCREEN_HEIGHT − 1，如下所示：

$d = (0.5)*(\text{SCREEN\_WIDHT}-1)*\tan(\theta_h/2)$

这样，如果将透视变换和屏幕变换合并时，可确保结果是正确的。然而，这里好像没有考虑 SCREEN_HEIGHT，事实并非如此。通常，我们首先将图像投影到视平面上，此时对于 x 和 y 轴，使用的视距相同，在视平面中，x 的范围为-1 到 1，y 的范围为$-1/ar$ 到 $1/ar$；然后将视平面映射到屏幕上：将 x 和 y 轴分别放大 SCREEN_WIDTH 和 SCREEN_HEIGHT 倍。但是，由于宽高比 ar 就是 SCREEN_WIDTH/SCREEN_HEIGHT，因此乘法和除法相互抵消掉了。因此，在不首先将相机坐标变换为视平面坐标，而直接将其映射到屏幕上得到视口屏幕坐标时，ar 项被消除了，结果如下：

$$x_{per} = (d*x_c) / z_c$$
$$y_{per} = (d*y_c) / z_c$$

结果已经被缩放到 SCREEN_WIDTH×SCREEN_HEIGHT，在不经过透视变换的情况下，直接完成了屏幕变换。然而，这种方法存在两个问题。在光栅显示器中，y 轴反转过来了，因为在光栅显示器中，原点位于左上角。这很简单，只需按下述方式将 y 反转即可：

$$y_{per} = (\text{SCREEN\_HEIGHT}-1) - y_{per}$$
$$x_{per} = x_{per}$$

第二个问题要复杂些，也更糟糕。虽然在 x 轴上，投影后 x 坐标的变化范围为 SCREEN_WIDTH－1，y 坐标的变化范围为 SCREEN_HEIGHT−1，但实际的坐标范围如下：

- x 轴：-（SCREEN_WIDTH－1）/2 到（SCREEN_WIDTH－1）/2
- y 轴：-（SCREEN_HEIGHT－1）/2 到（SCREEN_HEIGHT−1）/2

在每个轴上，坐标范围以零为中心，一半为正，一半为负。在归一化投影中，每个轴上的坐标范围为-1 到 1，这里的情况与此类似，只是坐标范围更大而已。虽然避免了从归一化的投影坐标变换为屏幕坐标的缩放操作，但仍然需要平移原点。确实是这样，但可以对这种情况和 y 轴反转进行处理，得到投影变换后的如下变换：

$$x_{screen} = x_{per} + (0.5*\text{SCREEN\_WIDTH}-0.5)$$
$$y_{screen} = -y_{per} + (0.5*\text{SCREEN\_HEIGHT}-0.5)$$

如果对上述操作进行分析，将发现它们与进行归一化投影但不执行缩放操作时相同。正如读者看到的，这些投影方式的结果相同，只是考虑问题的方式不同而已。事实上，如果像前面一样假定α= 0.5 *(SCREEN_WIDTH−0.5)，β= 0.5 *(SCREEN_HEIGHT−0.5)，则结果如下：

$$x_{screen} = x_{per} + \alpha$$
$$y_{screen} = -y_{per} + \beta$$

在矩阵中，只需要对(x_{per}, y_{per})进行平移即可，因此矩阵 $\mathbf{T}_{scr}$ 如下：

$$\mathbf{T}_{scr} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \alpha & \beta & 0 & 1 \end{bmatrix}$$

与如下所示的更复杂的归一化透视坐标到屏幕坐标变换相比，上述变换惟一的差别在于，没有将坐标 x_{per} 和 y_{per} 分别乘以缩放因子α和β而已。乘以缩放因子只是将视平面的大小放大到 2*α×2*β而已，因为在 x 轴和 y 轴上，实际的坐标范围分别为$-\alpha$到α和$-\beta$到β。

$$x_{screen} = \alpha + x_{per}*\alpha$$
$$y_{screen} = \beta - y_{per}*\beta$$

关键之处在于，可以在屏幕变换之前，在透视变换中对视平面坐标进行缩放（但得到的视平面坐标是非归一化的），也可以在屏幕变换期间，将归一化的视平面坐标进行缩放，得到屏幕坐标。

最后需要指出的是，我们介绍了两种执行透视变换的方式：归一化投影和将视平面的大小放大到与视口相同；同时介绍了两种相应的屏幕变换。还需要指出的一个细节是，在处理过程中将 4D 齐次坐标转换为 3D 坐标。我们将投影变换和屏幕变换分开，分别用一个矩阵表示，并在这两个变换之间，将每个顶点的齐次坐标转换为 3D 坐标，但如何通过手工或使用一个矩阵来一次性执行相机坐标到屏幕坐标变换呢？首先来看手工变换情况。

简单地说，需要一次性执行投影变换以及缩放和平移，其数学公式如下：

$$x_{screen} = d*x_{cam}/z_{cam} + (0.5*SCREEN\_WIDTH - 0.5)$$
$$y_{screen} = -d*y_{cam}/z_{cam} + (0.5*SCREEN\_HEIGHT - 0.5)$$

是不是很容易？正因为上述数学公式很简单，因此在很多引擎中，都不使用基于矩阵的透视变换和屏幕变换。有了顶点的相机坐标后，使用上述两个公式一次性将相机坐标变换为屏幕坐标。但如果要使用单个矩阵来表示这种变换，情况将复杂些，因为变换得到的坐标为 4D 齐次坐标，必须通过除以 w 分量将齐次坐标转换为 3D（实际上是 2D）坐标。看看下述变换矩阵是否正确：

$$T_{camscr} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & 1 & 1 \\ \alpha & \beta & 0 & 0 \end{bmatrix}$$

用相机坐标为[x_c y_c z_c 1]的顶点 p 来检验它：

$$p*T_{camscr} = [d*xc + \alpha,\ -d*y_c + \beta,\ z_c,\ z_c]$$

结果好像是对的。将其转换为 3D 坐标——将 x 和 y 分别除以 $w(z_c)$，结果如下：

$$x_{screen} = d*x_c/z_c + \alpha/z_c$$
$$y_{screen} = -d*x_c/z_c + \beta/z_c$$

将 4D 齐次坐标转换为 3D 坐标时，需要将每个分量除以 $w(z_c)$，这使得α和β项是错误的（都被除以 z_c）。因此，需要在原来的矩阵中将它们乘以 z_c，这样便能得到正确的结果。为此，将矩阵中的α和β向上移一行：

$$T_{camscr} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ \alpha & \beta & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

再次使用相机坐标为[x_c y_c z_c 1]的顶点 p 来检验它：

$$p*T_{camscr} = [d*x_c + z_c*\alpha,\ -d*y_c + z_c*\beta,\ z_c,\ z_c]$$

然后将坐标 x 和 y 分别除以 $w(z_c)$，结果如下：

$$x_{screen} = d*x_c/z_c + \alpha$$
$$y_{screen} = -d*x_c/z_c + \beta$$

警告：对于使用矩阵来执行透视变换和屏幕变换，需要注意的有两点：透视变换得到的是齐次坐标，必须将其转换为 3D 坐标；很多矩阵乘法运算和加法运算完全是浪费时间。

最后，读者可能会问，“宽高比怎么不见了呢”？原因在于将透视变换和屏幕变换合并起来后，宽高比被消除了。

那么，应该使用哪种投影方法呢？应该使用数学公式还是矩阵呢？本节提供了所有这些方法：归一化投影、将视平面的大小放大到与视口相同、数学公式和变换矩阵。在下一章的练习和演示程序中，我们将使用所有这些方法，但最终将选择视距为 1，x 轴的视野为 90 度的方式，因为作者喜欢使用归一化坐标。稍后我们将知道，编写代码时，将屏幕变换和透视变换合并起来，速度将更快。

另外，自由度为 3-4，可以设置视野、视距以及宽度和高度。然而，在大多数情况下，我们将 x 轴的视野固定为 90 度，并通过数学公式来计算 y 轴的视野，但无论如何，投影时对于两个轴应使用相同的视距。然而，对于非方形投影，必须考虑高宽比。

总之，读者可能认为，作者已经将整个投影过程阐述清楚了，但作者旨在让读者真正明白相机坐标、透视坐标、视平面、视口和屏幕坐标之间的关系，并了解使用各种变换得到这些坐标的所有方法，包括手工计算和使用矩阵。在本书中，大多数情况下我们都将水平视野设置为 90 度，将视距设置为 1.0。对于 y 轴，也将使用这样的视距值，因此对于非方形投影，将导致变形，但如果在透视计算中将 y 坐标乘以高宽比，则在屏幕变换中缩放 SCREEN_HEIGHT 时，将抵消这种变化。

3. 再看观察流水线

对几何体而言，此时的观察流水线是完整的。当然，我们还没有执行光照计算、着色处理和纹理

映射，但它们属于 3D 渲染的范畴，而现在我们只关心 3D 流水线中的几何变换部分。图 6.39 描述了现在的 3D 流水线。

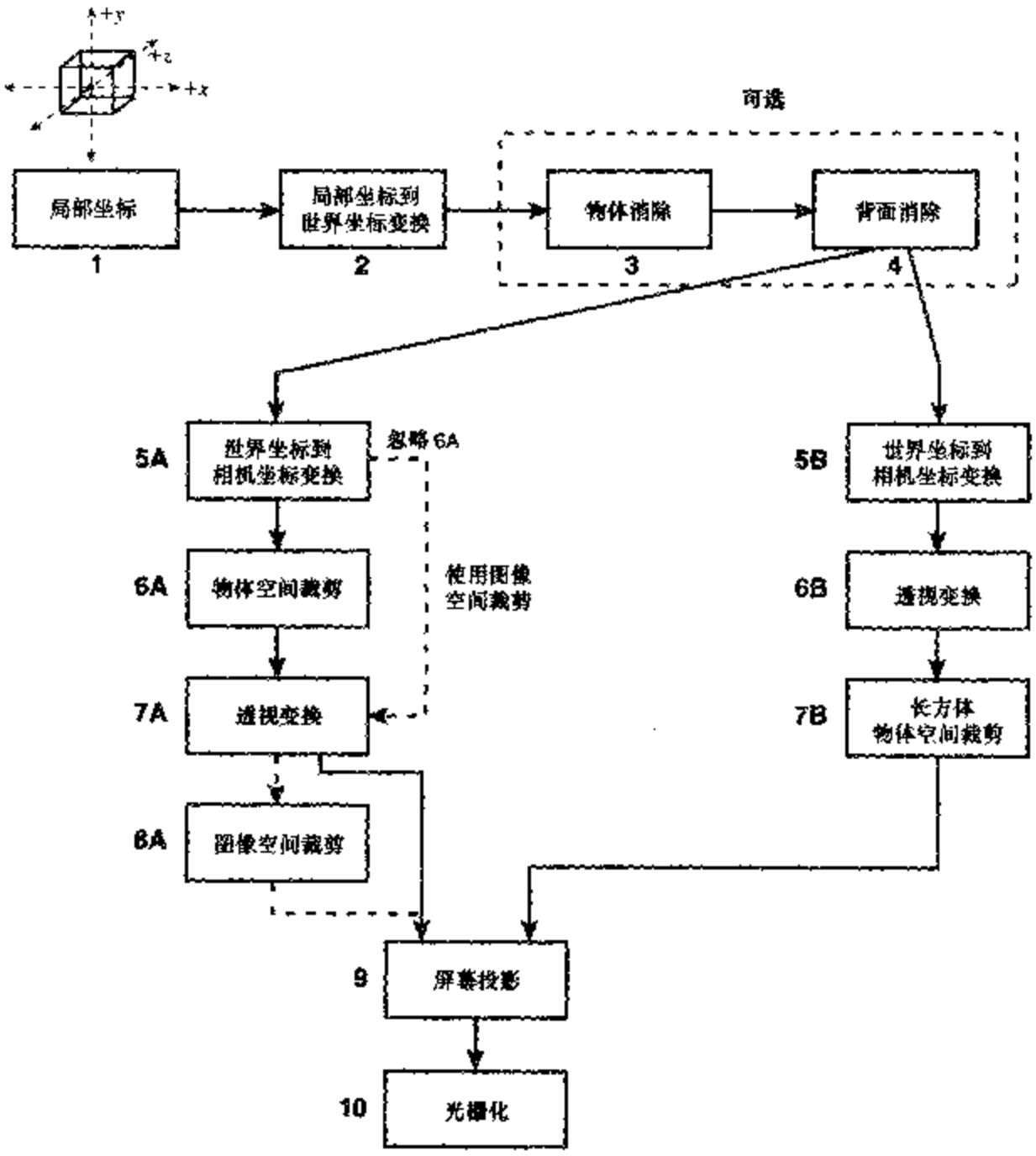


图 6.39　包含屏幕投影的 3D 流水线

6.4　基本的 3D 数据结构

现在，可以讨论通用的 3D 数据表示方法了。正如读者看到的，3D 数据将经过大量的变换，表示数据流的方法也有很多。因此，对于某种工作，没有放之四海皆准的处理方法，只有在特定条件下可行的方法。例如，存储太空游戏数据的方式可能与存储游戏数据的方式绝然不同。另外，根据裁剪图像的方式，在最终渲染的多边形中，可能需要插入或删除某些顶点，这要求预先考虑好。作者的朋友 John Amato 在编写其第一个 3D 引擎“SQL”时明白了这一点。

最起码，您需要对引擎深思熟虑，很多时候可能需要重做，因为您可能发现一种更佳的实现方式。本书采用的方法具有普适性，我们首先将使用一些粗糙的数据结构，然后在必要时对其进行调整。同样，要选择合适的数据结构，必须对 3D 的方方面面有深入认识，但是只有在读者阅读完本书后才能达到这样的程度，因此，现在我们将使用一些现成的东西，对问题进行简化，并做出一些假设。

作者的目标是，在本章结束前将创建出一个可行的 3D 线框引擎，让您能够在类似于地形图的环境中移动相机。该引擎将进行少量的裁剪，这种裁剪结合使用了物体空间裁剪算法和图像空间裁剪算法。接下来将讨论几种基本数据结构，用于在表示非常简单的通用 3D 系统中的 3D 物体。

6.4.1 表示 3D 多边形数据时需要考虑的问题

设计 3D 引擎时，首先需要做出的决策是，要支持三角形、四边形还是一般性 n 边多边形，如图 6.40 所示。大多数 3D 引擎（无论是基于软件还是硬件的）都只支持三角形。其原因很多，但最重要的原因是，三角形易于渲染——3 个点定义一个平面，同时很多 3D 算法最适合用于三角形网格。

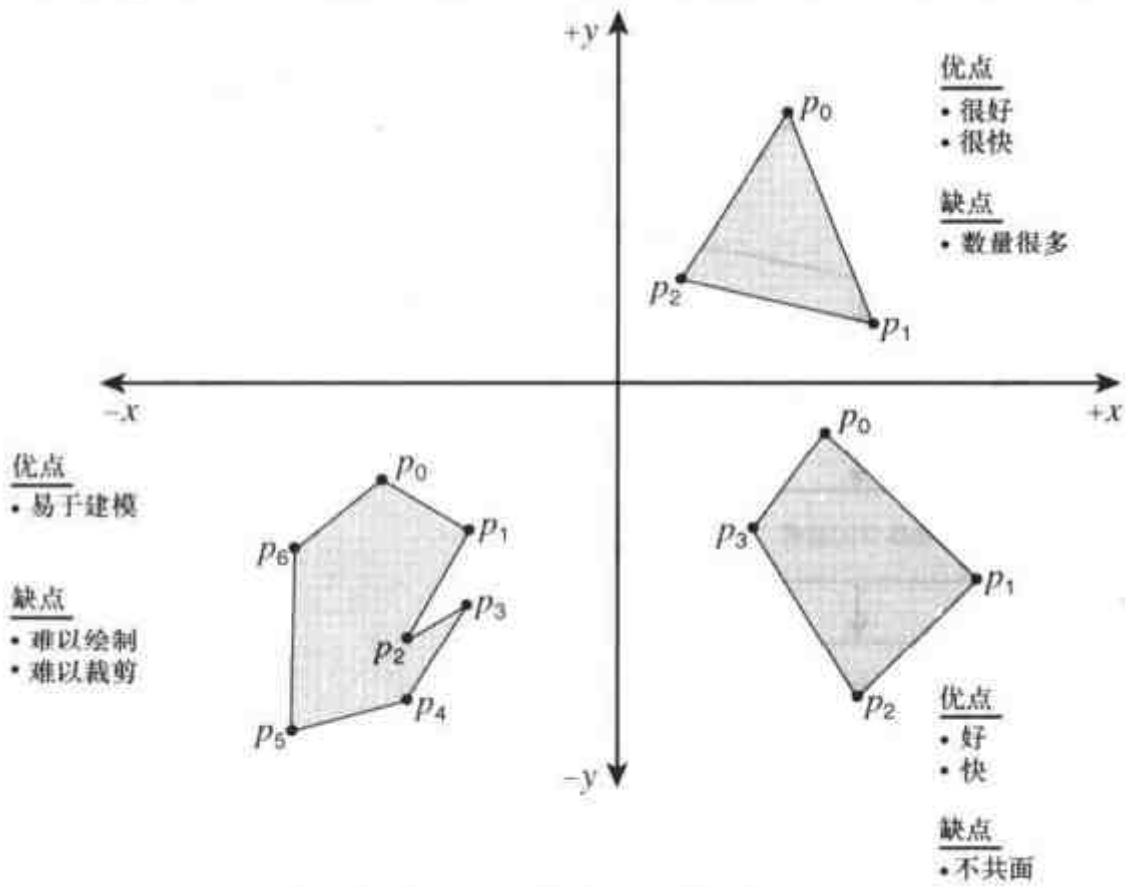


图 6.40 3D 引擎可能使用的多边形图元

当然，在建模工具中建立的模型可能包含一般性 n 边多边形，但最终所有模型都将用三角形表示。由于标准做法是使用三角形，因此我们将遵循这种约定，规定所有模型都必须是由三角形组成的，否则将在渲染之间将其转换为三角形。现在，我们还不知道如何获得物体的数据。是使用算法来生成、手工输入还是从文件中加载？这个无关紧要，我们只知道要处理的是基于三角形的几何体。

即使有了上述约束条件，仍然还有一个问题。在图 6.41 中，使用一个矩形来裁剪几个三角形。在裁剪阶段，可能会增加多边形的顶点数。例如，在最糟糕的情况下，三角形被裁剪后，可能增加 3 个顶点。因此，数据结构必须能够应对这种情况。

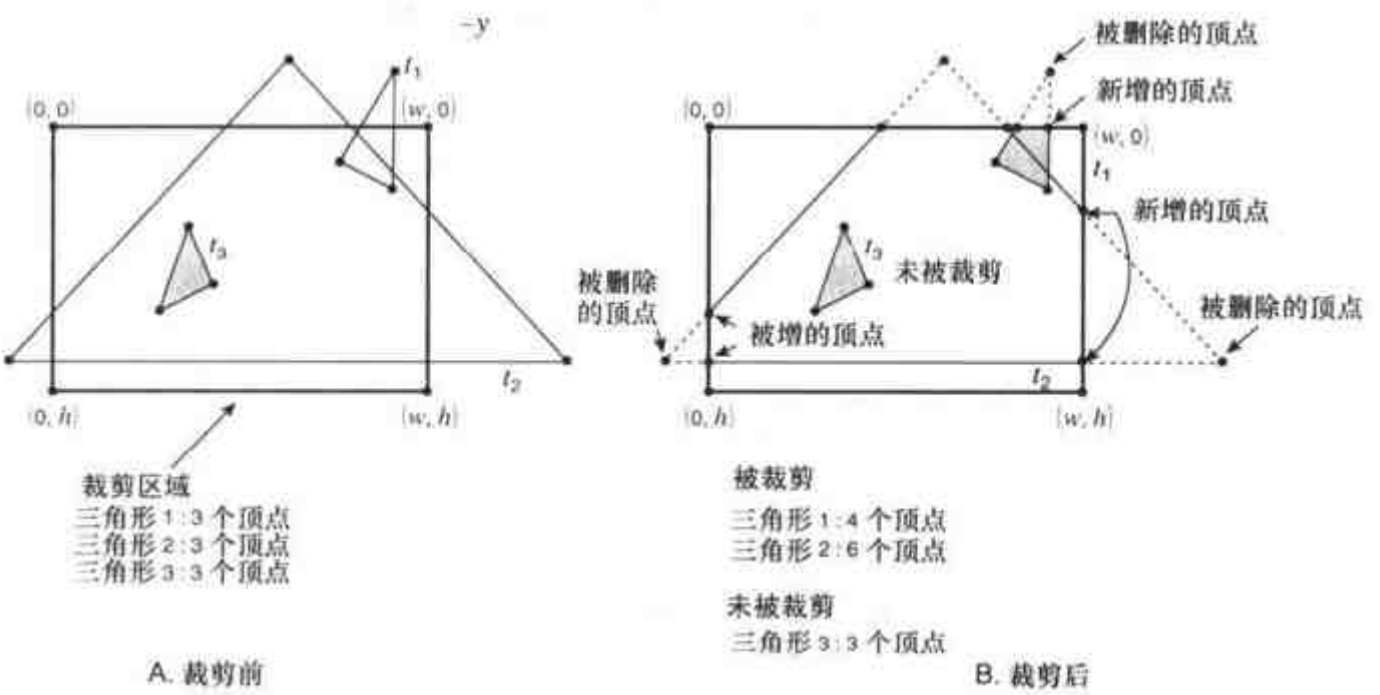


图 6.41 在物体空间中裁剪三角形新增顶点

读者可能会说，可以不在物体空间进行裁剪，而是等到渲染时再裁剪。这很好，但是多边形可能跨过

近裁剪面，甚至更糟的是跨围 $z = 0$ 平面，如图 6.42 所示。对这些多边形进行投影时将出现问题，这一点必须考虑。当然，如果所有多边形都很小，最长的边短于近裁剪面和平面 z = 0 之间的距离，则可以很容易地剔除位于完全位于该范围内的多边形，避免对其进行裁剪，然后在光栅化期间对像素执行图像空间裁剪。

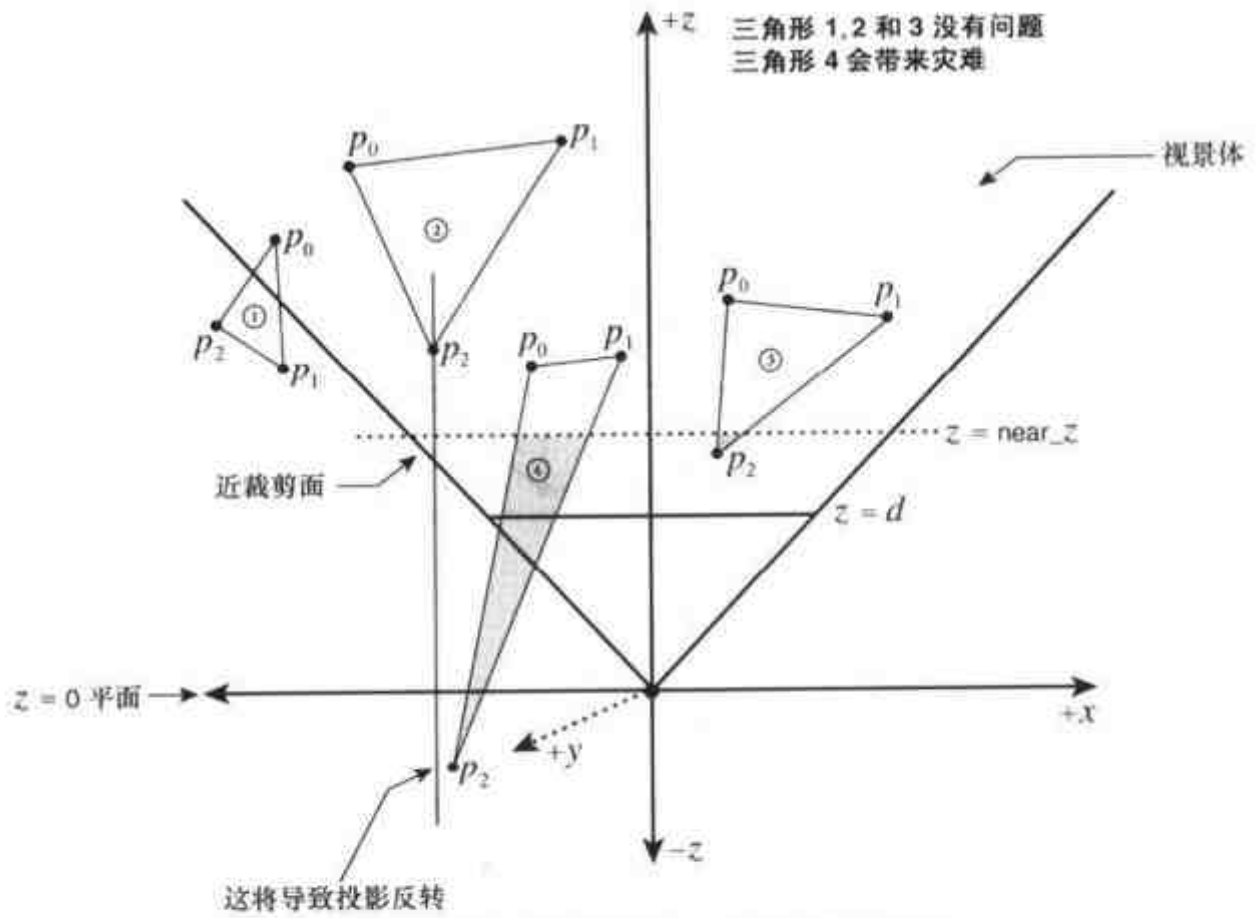


图 6.42　穿过近裁剪面和 z = 0 平面的三角形

基本原则是，创建数据结构时，必须根据将使用的几何体类型和裁剪方法，做出前瞻性和深思熟虑的考虑。这些因素将影响多边形的表示方法，如果多边形的顶点可以超过 3 个，它们将是决定性因素。

读者可能会说，可以规定所有模型都必须由三角形组成，如果经 3D 流水线的裁剪阶段后，某个多边形的顶点超过 3 个，可以将其当成特例来处理：将其划分成三角形，或对于 n 边多边形使用一个特殊的光栅化器。这种方法是可行的，作者也曾使用过。总之，必须在某个时候考虑所有这些因素，但现在为简单起见，暂时不管它们。

6.4.2　定义多边形

为打响多边形定义战役的第一枪，我们使用三角形规则，并假定在世界空间进行裁剪时，不会出现增加顶点的情形。也就是说，如果在世界空间或相继空间进行裁剪，要么将整个多边形剔除，要么保留它，让光栅化器在光栅化阶段根据屏幕对多边形进行裁剪（在我们的第一个引擎中，这非常简单，但这个引擎是线框型的）。这种简化使得我们在本章中无需涉及 3D 裁剪算法，这一主题将在以后介绍。定义多边形时，首先需要就 3D 世界的某些方面做出决策，例如要建立什么模型？

例如，假设要建立一个 3D 立方体模型，如图 6.43 所示。立方体有 6 个面，每个面由两个三角形构成（别忘了，不能使用四边形）。总共有 12 个多边形和 36 个顶点。是这样吗？确实有 12 个多边形，但总共 36 个顶点还是 8 个顶点呢？答案取决于如何存储模型。如果将每个多边形都视为一个独立的实体，将使用 3 个顶点来定义它：

```
struct typedef POLY_EX_TYP_1
{
POINT3D v[3]; // 顶点列表
} POLY_EX_1, *POLY_EX_1_PTR;
```

对于任何多边形，我们都将采用类似于下面的方式来定义：

```
POLY_EX_1 face_1 = { {x0, y0, z0},
                     {x1, y1, z1},
                     {x2, y2, z2}, };
```

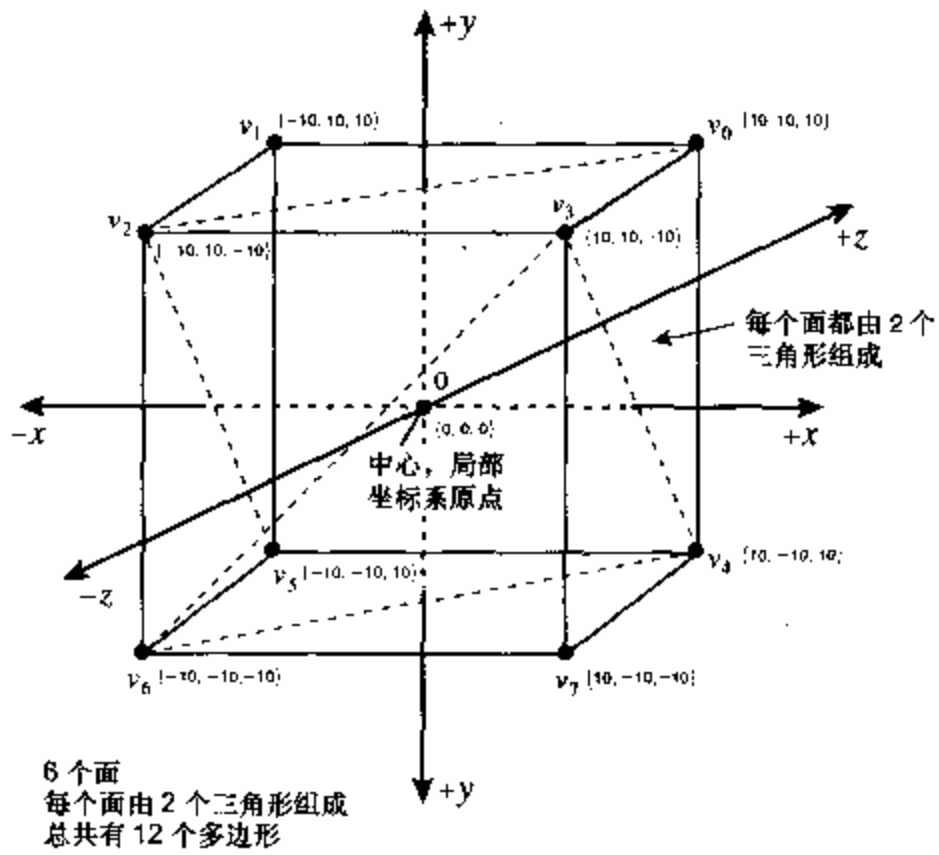


图 6.43　立方体模型

这很好，也可行。但这样做浪费了存储空间，因为立方体总共只有 8 个顶点，如果我们采用上述方式定义立方体，将有 12 个多边形，每个多边形有 3 个顶点。应该能够更好地重用顶点数据，为此，可以采用一种间接方法，基于顶点列表来定义多边形。

顶点列表是一组用来定义几何体的顶点，如图 6.44 所示。因此，可以使用一个数组来定义立方体的顶点，然后使用指向该数组的指针或引用来定多边形。例如，可以这样定义顶点列表：

```
POINT3D vertex_list[NUM_VERTICES]; // 顶点列表
```

然后使用立方体 8 个顶点的坐标来初始化数组的元素：

```
vertex_list[0] = {x0, y0, z0};
vertex_list[1] = {x1, y1, z1};
vertex_list[2] = {x2, y2, z2};
vertex_list[3] = {x3, y3, z3};
vertex_list[4] = {x4, y4, z4};
vertex_list[5] = {x5, y5, z5};
vertex_list[6] = {x6, y6, z6};
vertex_list[7] = {x7, y7, z7};
```

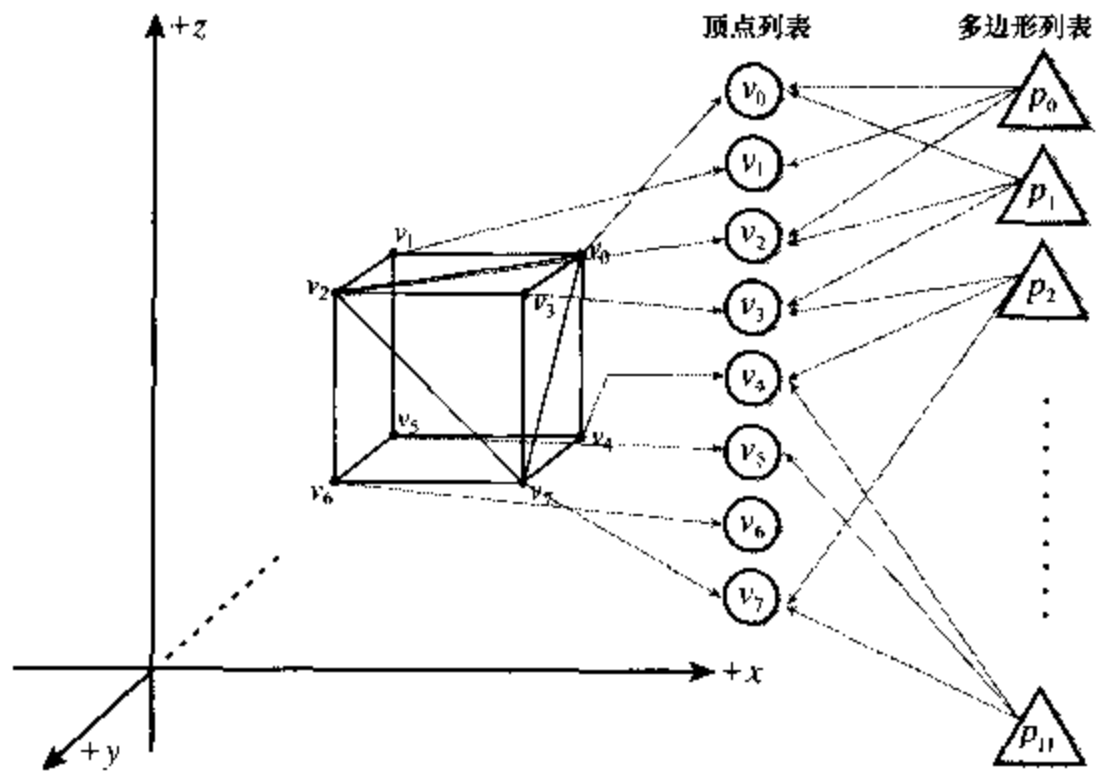


图 6.44　可以使用顶点列表来定义几何体

接下来定义一个新的多边形类型，使用顶点列表来生成多边形：

```
struct typedef POLY_EX_TYP_2
{
POINT3D_PTR vlist; // 顶点列表本身
int vertices[3];   // 指向顶点列表的索引

} POLY_EX_2, *POLY_EX_2_PTR;
```

现在可以使用间接方法了。在每个多边形中，引用同一个顶点列表，而不是在多边形中重复存储顶点而浪费空间。下面的代码定义了一个多边形：

```
POLY_EX_2 face_1 = {vertex_list, 0, 1, 2};
```

上述代码指出，多边形 face_1 由顶点数组 vertex_list 中的 3 个顶点（0、1 和 2）组成，这 3 个顶点的坐标分别是(*x*0，*y*0，*z*0)、(*x*1，*y*1，*z*1)和(*x*2，*y*2，*z*2)。

进行实体建模时，还有另一个细节非常重要，这就是顶点顺序。也就是说，定义多边形或三角形时，需要以特定的方式排列顶点，这样才能确定哪一面是外面。

在图 6.45 中，三角形的顶点是按顺时针方向排列的。还记得背面消除吗？要进行背面消除，首先需要确定哪一边是前面，哪一边是后面，然后计算一条法线（当然，如果从两边看，多边形都可见，这就无关紧要）。为此，需要使用一种顶点顺序方案，这样才能以统一的方式计算指向外面的法线。

如果使用左手坐标系和顺时针顶点排列方案，可以使用左手规则来确定向外的方向：手掌环绕方向与顶点编号递增的方向相同时，拇指指向的方向就是向外的。换句话说，如果我们创建向量 u = **p0->p1** 和 v = **p0->p2**，然后计算向量 u 和 v 的叉积，得到的将是法线 n，如图 6.45 所示。当然，这就是外面的方向。进行背面消除时，我们使用的就是向量 n。在本书中，总是使用顺时针顶点顺序，来定义所有多边形的向外方向。然而，这种顶点顺序方案只适用于左手坐标系。在右手坐标系中，必须反转法线的方向（负号）或使用右手规则。

提示：对于任何 n 边多边形，都有 n 种不同的顺时针（逆时针）顶点排列顺序，也就是说，排列顺序不是惟一的。在很多情况下，如果以编号从小到大的顺序排列顶点，将可提高数据连贯性（data coherence）。例如，假设多边形顶点排列顺序 0-5-12 是顺时针的，就应该采用这种顺序，虽然 12-0-5 和 5-12-0 也是顺时针的，但在后面两种情况下，在顶点数组中存取数据顺序是颠倒的，因此在大多数情况下第一种排列顺序的存取性能更高。然而，在很多情况下，存取最后一个元素，强制更新缓存，然后再存取第一个元素的速度更快。

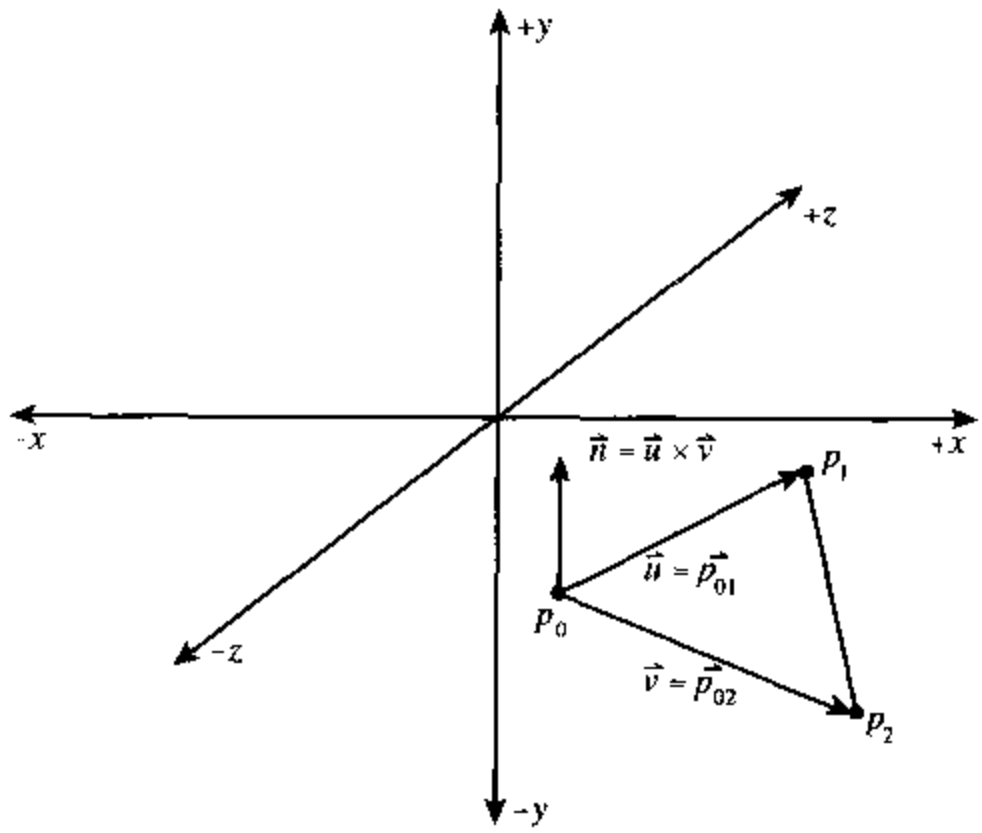


图 6.45　顶点顺序

回到顶点列表。顶点列表是一种不错的表示方法，使用这种方法有很多好处。例如，要对物体进行变换，无需对组成物体的多边形进行变换，而只需对顶点列表进行变换即可。另外，有表示同一个顶点的多

个拷贝时，便可实现数值漂移（numerical drift）。也就是说，如果有两个多边形共享同一个顶点，但每个多边形都有这个顶点的拷贝，然后分别对这两个多边形及其顶点拷贝进行变换时，但在变换期间如果由于某种原因导致某个拷贝出现细微的误差，这两个原来被视为同一个顶点的顶点将不再是这样的。这可能导致几何体出现裂缝。只要看看 Tomb Raider，就能明白这里讨论的问题。

下面来看看使用顶点列表的缺点。很多情况下，随着您沿着 3D 流水线往下走，可能想将多边形和顶点列表分开。其原因很多，例如，假设有一个包含 1000 个顶点的网格，而只有一个多边形是可见的。由于所有的变换都是在顶点列表级进行的，因此将对所有的数据进行变换，而目的只为变换一个多边形。当然，有办法避免这种情况发生，但这毕竟是个问题，有必要注意。另外，在很多时候，编写根据顶点列表中的 3 个顶点（而不是一系列指向顶点列表的指针）来渲染多边形的光栅化器更容易。同样，所有上述因素都取决于最后的体系结构。

最后，在裁剪期间，如果几何体被表示为顶点列表，而不将其视为多边形，将很难进行裁剪。同样，如果裁剪一个顶点，所有共享该顶点的多边形都将被裁剪；至此，读者应该明白，需要考虑的问题很多。

因此，我们将使用混合方法，基于顶点列表来定义和变换物体；然而，在某个时候需要将它们转换为多边形，即使这些多边形共享顶点。这种转换将在以后介绍，现在暂时不去管它。

在定义第一个多边形结构之前，有必要对另一个之前被忽略的细节进行说明：使用 3D 坐标还是 4D 齐次坐标。这是一个棘手的问题，正确的做法是两种都使用。我们最终将使用 3D 坐标，并假设分量 $w=1$，并使用之前编写的 4×3 矩阵乘法代码。但现在我们应使用 4D 齐次坐标和 4×4 矩阵，以便接下来的两章可以以统一的方式介绍使用矩阵来执行裁剪、透视和投影。虽然我们知道，我们并不需要 4D 齐次坐标提供的 $w=1$，而可以在代码中假设 $w=1$，不过现在我并不想做这样的假设。为确保数学公式和矩阵运算的一致性，我们假设在大多数情况下，所有的点和顶点的 4D 齐次坐标中 $w=1$。接下来将基于上述约定，创建一个更可靠的顶点列表和一种多边形数据结构。

第一个多边形

我们忽略了很多有关多边形的细节，因为我们现在对渲染还不太了解，无法考虑这些细节。然而，在下一章中设计线框引擎时，至少需要多边形的下述信息：

- 颜色——多边形的颜色。这可以是 8 位的颜色索引，16 或 24 位的 RGB 值，也可以是您虚构的其他东西。
- 状态——多边形的状态。用于存储在多边形被处理的过程中动态变化的状态信息。例如，多边形是否处于活动状态（active）、是否被裁剪以及是否受损等。
- 属性——多边形的物理属性，可能用于存储多个描述多边形的标记，例如是否是双面的、是否反射光、是不是由钢材制成、是否透明等。

根据约定，我们假设多边形的顶点数不超过 3 个（就现在而言）。下面是我们创建的第一个多边形结构版本：

```
typedef struct POLY4DV1_TYP
{
int state;    // 状态信息
int attr;     // 多边形的物理属性
int color;    // 多边形的颜色

POINT4D_PTR vlist; // 顶点列表
int vert[3];       // 顶点列表元素的索引

} POLY4DV1, *POLY4DV1_PTR;
```

注意：这里使用的命名规则是：[root][coordinate type][version]。POLY 是物体类型，4D 是坐标类型，V1 为版本号。

vlist 指向的顶点列表的类型为 POINT4D[ ]。为确保 state、attr 和 color 的抽象性，避免作茧自缚，我们不打算进一步定义它们。下面创建另一个数据结构，作者将其称为“面（face）”。

还记得吗，在光栅化阶段，需要一个独立于物体（顶点列表）的多边形列表。也就是说，需要一个多边形面数组或链表，其中每个多边形面都是自包含的，可以将其传递给光栅化器。为此，接下来创建一种名为 POLYF4DV1 的数据结构，用于存储多边形面。除了是自包含的（即没有引用外部顶点列表）外，它与数据结构 POLY4DV1 相同：

```
typedef struct POLYF4DV1_TYP
{
int state;    // 状态信息
int attr;     // 多边形的物理属性
int color;    // 多边形的颜色

POINT4D vlist[3];  // 三角形的顶点
POINT4D tvlist[3]; // 变换后的顶点
POLYF4DV1_TYP *next; // 指向列表中下一个多边形的指针
POLYF4DV1_TYP *prev; // 指向列表中前一个多边形的指针

} POLYF4DV1, *POLYF4DV1_PTR;
```

这个数据结构中包含指针 next 和 prev，它们用于建立如图 6.46 所示的多边形链表，这种链表可被传递给光栅化器。可能需要这两个指针，也可能不需要，这里定义它们旨在预先做好准备。另外，还有两个顶点数组：vlist[ ]和 tvlist[ ]，后者用于存储对前者进行变换得到的结果，这样可避免操纵多边形时覆盖原来的顶点数据。同样，数组 tvlist[ ]可能用得上，也可能用不上。以后，可能还需要添加用于存储世界坐标、相机坐标、透视坐标和屏幕坐标的顶点数组。

数据结构 Poly F4DV1

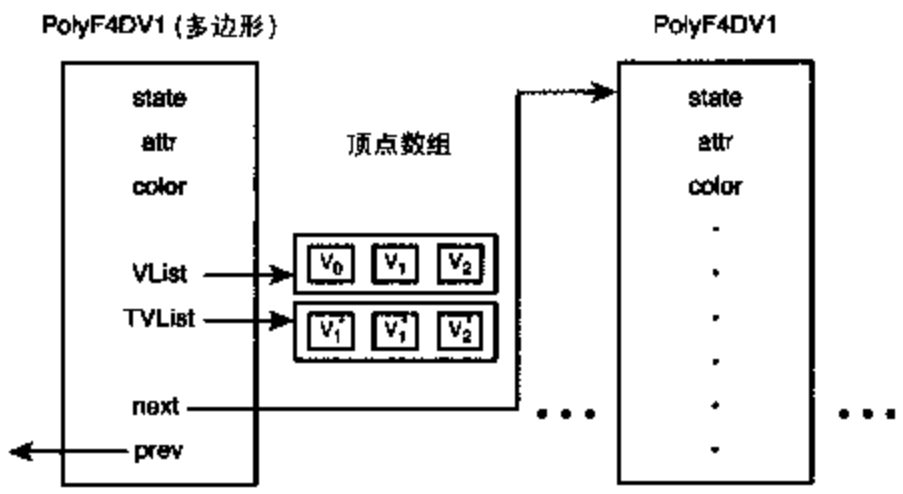


图 6.46　数据结构 POLYF4DV1

注意：不能删除最初的 3D 模型局部坐标。要么复制它们，要么在变换时将结果存储到另一个数据结构中，因为它们是一切数据的最终来源。另一方面，如果在局部坐标空间中执行了诸如旋转和缩放等变换，则可以将原来的顶点数据删除，因为您希望模型始终保持执行局部变换后的状态。

至此，我们创建了顶点列表、多边形数据结构以及可用于渲染的多边形面数据结构，具备了最基本的条件。接下来将定义物体，再增加一个层次。

6.4.3　定义物体

在 3D 游戏中，物体通常是由一组多边形构成的，如图 6.47 所示。然而，在较高级的 3D 系统中，物体可能是由多个通过铰链（link point）相连的物体组成的，如图 6.48 所示。这种物体将在后面讨论，现在只处理如图 6.47 所示的简单物体，这种物体由刚性多边形网格构成，没有活动部件。在这种情况下，物体由一组多边形组成，物体本身是基于一个顶点列表的。基于这种抽象模型，可以轻松地创建出一种存储物体的数据结构。然而，对于与 3D 流水线相关的数据存储方式，现在必须做出决定。

有局部坐标、世界坐标、相机坐标、透视坐标和屏幕坐标。需要为所有这些坐标提供存储空间吗？答案取决于很多因素，但至少需要为局部坐标提供存储空间。至于是否需要为其他坐标提供辅助存储空间，完全取决于 3D 流水线的内部工作方式。也就是说，将物体传递给流水线时，由物体本身还是流水线负责提供辅助存储空间。

这是一个棘手的问题，没有放之四海皆准的答案。作者采用的方法是，让物体本身提供存储局部坐标、世界坐标和相机坐标的空间，同时在观察流水线中提供存储这些坐标的功能。

就现在而言，处于简化的目的，将物体视为包容一切（包括流水线的变换结果）的容器。然而，我们只提供存储局部坐标和变换后坐标的空间。变换后的坐标可以是世界坐标、相机坐标、透视坐标或屏幕坐

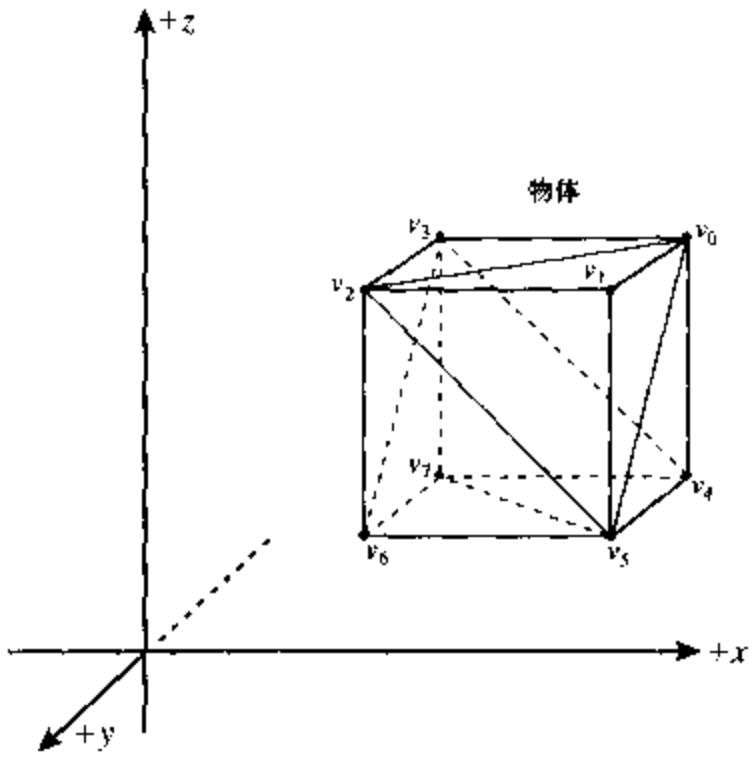


图 6.47　由多边形构成的物体

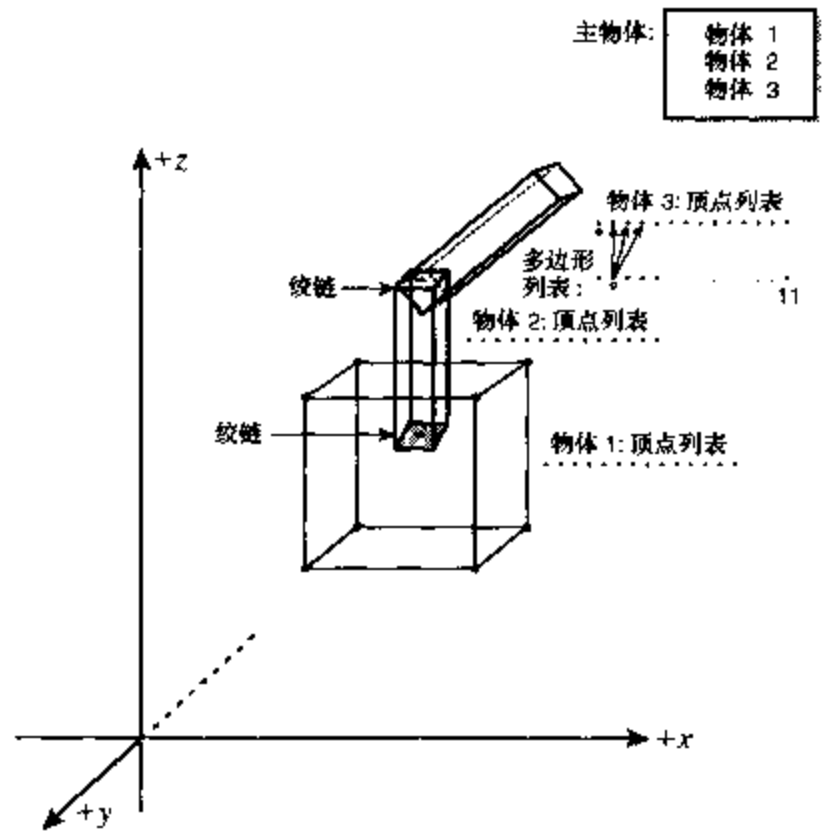


图 6.48　由多个通过铰链相连的物体组成的物体

标：只能通过数据处于流水线的哪个阶段来确定变换后的坐标是哪种坐标。这最大限度地减少了所需的存储空间。在物体数据结构中，还需要定义哪些数据呢？需要定义顶点数、属性、状态和物体名称（或 ID），以及后面需要使用的包围半径（bounding radius）。我们的第一个物体数据结构版本如下：

```
// 基于顶点列表和多边形列表的物体
typedef struct OBJECT4DV1_TYP
{
int  id;              // 物体的数值 ID
char name[64];        // 物体的字符串名称
int  state;           // 物体的状态
int  attr;            // 物体的属性
float avg_radius;     // 物体的平均半径，用于碰撞检测
float max_radius;     // 物体的最大半径

POINT4D world_pos;    // 物体在世界坐标系中的位置

VECTOR4D dir;         // 物体在局部坐标系下的旋转角度
                      // 为用户定义的坐标或单位方向向量

VECTOR4D ux,uy,uz;    // 记录物体朝向的局部坐标轴
                      // 物体被旋转时，将相应地旋转

int num_vertices;     // 物体的顶点数

POINT4D vlist_local[64]; // 存储顶点局部坐标的数组
POINT4D vlist_trans[64]; // 存储顶点变换后坐标的数组

int num_polys;          // 物体的多边形数
POLY4DV1 plist[128];  // 存储多边形的数组

} OBJECT4DV1, *OBJECT4DV1_PTR;
```

对于上述数据结构 OBJECT4DV1，读者的第一印象是，它包含的静态数据会浪费大量的存储空间。这不是作者的疏忽，而是故意为之，旨在避开动态分配/释放内存的问题。以后的版本将动态地为顶点列表和多边形列表分配内存。就现在而言，64 个顶点和 128 个多边形足以描述立方体、岩石和玩具。另外，这个数据结构中还包含存储物体最大半径和平均半径的字段，它们用于剔除物体。

当然，还有存储物体位置的变量 world_pos 和存储物体在局部空间中朝向的变量 dir。在局部坐标系中对物体的朝向进行变换时，将修改这些局部坐标，因此需要采取某种方法来记录这些修改。当然，也可以像在流水线中对物体进行处理时那样，动态地执行所有的局部旋转，并保留原来的朝向数据，但如果沙漠

中有 1000 块不需要旋转的岩石，这样做将浪费时间。在很多情况下，3D 引擎在物体的局部空间中对其进行旋转，并覆盖原来的局部坐标，但需要记录旋转结果，这就是变量 dir 的用途。

可以使用该变量来存储角度或一个单位向量，其初始值与 z 轴平行（dir=<0，0，1>），并在物体每次被旋转时相应地修改其值，这样便可以记录物体被加载时的朝向。另外，向量 u_x、u_y 和 u_z 也记录物体的朝向，即每当物体被变换时，u_x、u_y 和 u_z 也相应地被变换，从而记录物体变换后的朝向。例如，假设要基于下述顶点列表定义一个立方体：

```
p0(10,10,10)
p3(-10,10,10)
p2(-10,10,-10)
p1(10,10,-10)
p4(10,-10,10)
p5(-10,-10,10)
p6(-10,-10,-10)
p7(10,-10,-10)
```

代码如下：

```
// 手工定义立方体（极其繁琐）
OBJECT3DV1 cube_1;

cube_1.id    = 0;              // 当前未使用
cube_1.state = 0;              // 当前未使用
cube_1.attr  = 0;              // 当前未使用

cube_1.avg_radius  = 17.3;     // 物体的平均半径
cube_1.max_radius  = 17.3;     // 物体的最大半径
                               // sqrt(10*10+10*10+10*10)


cube_1.world_pos.x = 0; // 物体在世界坐标系中的位置
cube_1.world_pos.y = 0;
cube_1.world_pos.z = 0;

cube_1.local_ang.x = 0; // 物体的朝向
cube_1.local_ang.y = 0; // 这是所有物体被加载时使用的默认朝向
cube_1.local_ang.z = 1;
cube_1.num_vertices =  8; // 顶点数

// 创建一个顶点列表以简化初始化工作
POINT3D temp_verts[8] = { {10,10,10},     // p0
                          {10,10,-10},    // p1
                          {-10,10,-10},   // p2
                          {-10,10,10},    // p3
                          {10,-10,10},    // p4
                          {-10,-10,10},   // p5
                          {-10,-10,-10},  // p6
                          {10,-10,-10}};  // p7
// 设置顶点的局部坐标

for (int vertex=0; vertex < 8; vertex++)
    {
    cube_1.vlist_local[vertex].x = temp_verts[vertex].x;
    cube_1.vlist_local[vertex].y = temp_verts[vertex].y;
    cube_1.vlist_local[vertex].z = temp_verts[vertex].z;
    cube_1.vlist_local[vertex].w = 1;
    } // end for index

cube_1.num_polys = 12; // 多边形数

// 接下来定义多边形
// 这里将 state、attr 和 color 都设置为 0
// 将顶点指针指向局部坐标数组，但将其指向哪个列表无关紧要
// 因为所有的多边形都基于自己的顶点列表，而不是外部顶点列表
```

```
// 创建一个定义了每个多边形的三角形列表以简化初始化工作
// 所有三角形的顶点都按顺时针方向排列，因此方向朝外
int temp_poly_indices[12*3] = {
0,1,2, 0,2,3,  // 多边形 0 和 1
0,7,1, 0,4,7,  // 多边形 2 和 3
1,7,6, 1,6,2,  // 多边形 4 和 5
2,6,5, 2,3,5,  // 多边形 6 和 7
0,5,4, 0,3,5,  // 多边形 8 和 9
5,6,7, 4,5,7 };// 多边形 10 和 11

// 下面初始化每个三角形

for (int tri=0; tri < 12; tri++)
    {
    // 这些变量当前都未使用
    cube_1.plist[tri].state = 0;
    cube_1.plist[tri].attr  = 0;
    cube_1.plist[tri].color = 0;

    // 将顶点列表指向局部坐标数组
    cube_1.plist[tri].vlist = cube_1.vlist_local;

    // 设置顶点索引
    cube_1.plist[tri].vert[0] = temp_poly_indices[tri*3+0];
    cube_1.plist[tri].vert[1] = temp_poly_indices[tri*3+1];
    cube_1.plist[tri].vert[2] = temp_poly_indices[tri*3+2];
    } // end for
```

定义一个简单的立方体，竟然要这么多代码。这是 3D 图形最重要的问题之一：需要大量的数据。编写游戏时，不可能手工输入表示位图的数据；同样，在编写 3D 游戏时，也不可能手工输入表示物体和网格的数据，而总是从文件中加载网格和游戏背景，因为它们包含的顶点有数百、数千甚至数百万个。在 3D 物体过于复杂之前，我们将通过手工输入，使用 ASCII 文件格式来表示 3D 物体。

定义立方体后，可以将其传递给 3D 流水线，后者能够设置相机位置和视景体，执行前面介绍的所有操作。最后，将以填充多边形或线框模式渲染物体，但这是下一章将介绍的内容。

6.4.4 表示世界

至此，我们的 3D 世界由 3D 物体组成，每个物体都是一组基于顶点列表的多边形。然而，如果要编写类似于 Quake 那样的游戏，或在非常广阔的地形中模拟坦克，该如何办呢？我们的物体数据结构完全可用于定义物体游戏中的物体，但无法定义游戏 Quake 中的室内环境和坦克模拟游戏中的大型室外地形图。

室内和室外世界（统称为环境）通常是特殊的物体，由于数量庞大，必须使用更复杂的数据结构来表示。例如，游戏中的坦克或角色通常由 20～500 个多边形组成；而游戏世界可能由数万甚至数千万个多边形组成。不能在一个大型列表中存储这些多边形，并将其传递给 3D 流水线，而需要使用一个更灵巧的数据加载方法，因为在任何时刻，都只有一部分多边形是可见的。在本章前面，我们讨论了一些可用于表示地形的方法，其中一种解决方案是将世界表示为一系列的区段（sector），也许区段就是物体？在另一方面，前面介绍的有些算法（如物体消除）不再管用，因为涉及地形或室为环境时，没有物体的概念，取而代之的是房间、区域和入口（portal）等，我们使用类似的方法来剔除它们。

另外，大多时候地形和室内环境变化不大，因此可以预先进行大量的计算和优化，并利用二叉空间分割（binary space partition，BSP）、八叉树和入口来进行简化，并最大限度地减少在每一帧中需要处理的数据量。例如，由于环境不会移动，因此可以使用世界坐标来定义，从而避免局部坐标到世界坐标的变换。

因此，在大多数情况下，确定环境的哪些部分是可见的后，只需使用相机变换矩阵和投影矩阵对其进行变换。

本书后面将讨论所有这些概念，但需要指出的是，使用现在介绍过的技术，完全可以编写出以线框和恒定着色处理模式渲染的 3D 游戏。要编写更复杂的游戏，必须创建出强大得多的数据结构和算法，以减少每帧中需要处理的数据量。

因此，没有理由现在就试图去创建用于表示世界的 3D 数据结构，因为我们的 3D 图形知识还不够；实

际上，我们需要根据游戏世界来调整设计方案。然而，我们可以想象一下，对于类似于 Quake 的游戏，可能设计这样的引擎：将每个房间作为一个独立的树节点；每个房间都有指向从该房间中能够看到的其他房间的指针。使用这种数据结构，在任何时候都能够根据当前的位置，将除少数几个房间之外的其他所有房间剔除，如图 6.49 所示。

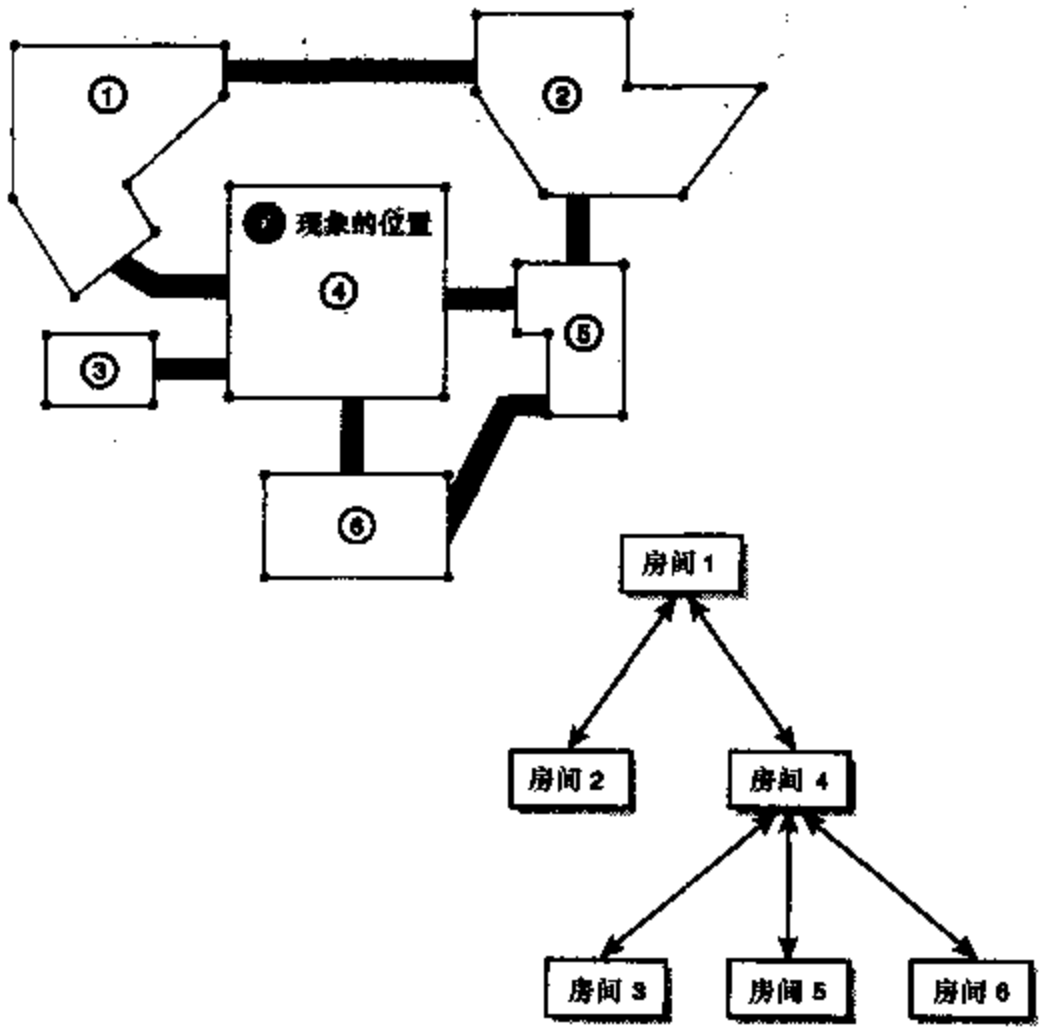


图 6.49　一种可行的数据结构

当然，如果从房间中可以看到外面或房间有很多窗户，这种数据结构将不管用。如果在您编写的游戏中，所有房间都通过走廊连接在一起，总共有 1000 万个房间，引擎将能够迅速将它们剔除。这是 BSP 树、八叉树和入口基于的一些思想。

6.5　3D 工具

当前，制作 3D 游戏的工作量非常大，主要原因是游戏是 3D 的，需要的数据非常多。游戏程序员/设计人员面临的最苦恼的问题之一是，如何生成和处理游戏数据。这个问题的解决方案有很多。对于简单的 3D 游戏，可以自己创建用于 3D 建模和生成世界的工具。例如图 6.50 是 DoomEdit 的一个屏幕截图。正如读者看到的，这是一个基于位图和直线的工具，显示的是俯视图。这个工具能够读取各种数据文件，让关卡设计人员能够加入物体、几何体、插销（switch）、动物和光源等，以创建游戏世界。DoomEdit 和其他“构建引擎（build engine）”用于生成整个 3D 游戏世界，是一种全方位（all in one）解决方案。当然，诸如脚本、位图和声音等游戏资源（game asset）是使用其他工具创建的。

当前，游戏越来越复杂，因为游戏世界是 3D 的，其中的几何体形状是任意的。因此，可采取的方法有两种，一是仍然自己编写 3D 工具，但最终您将发现自己实际上是在编写一个 3D 建模软件包；另一种方法是使用现成的 3D 建模程序，如 Caligari trueSpace、Maya、LightWave、3D Studio Max、Blender 或 Moray（和光线跟踪程序 Persistence of Vision），来设计物体和游戏世界，然后通过软件或插件在这些工具和游戏引擎之间连接起来。这可能是最佳的方法，因为您利用了他人所做的工作。在另一方面，3D 建模程序不擅长于加入怪物（monster）、大门插销等物体，不过功能在不断完善，很多建模程序提供了大量的挂钩（hook）和 API，让您能够对所需的建模程序基本功能进行修改。

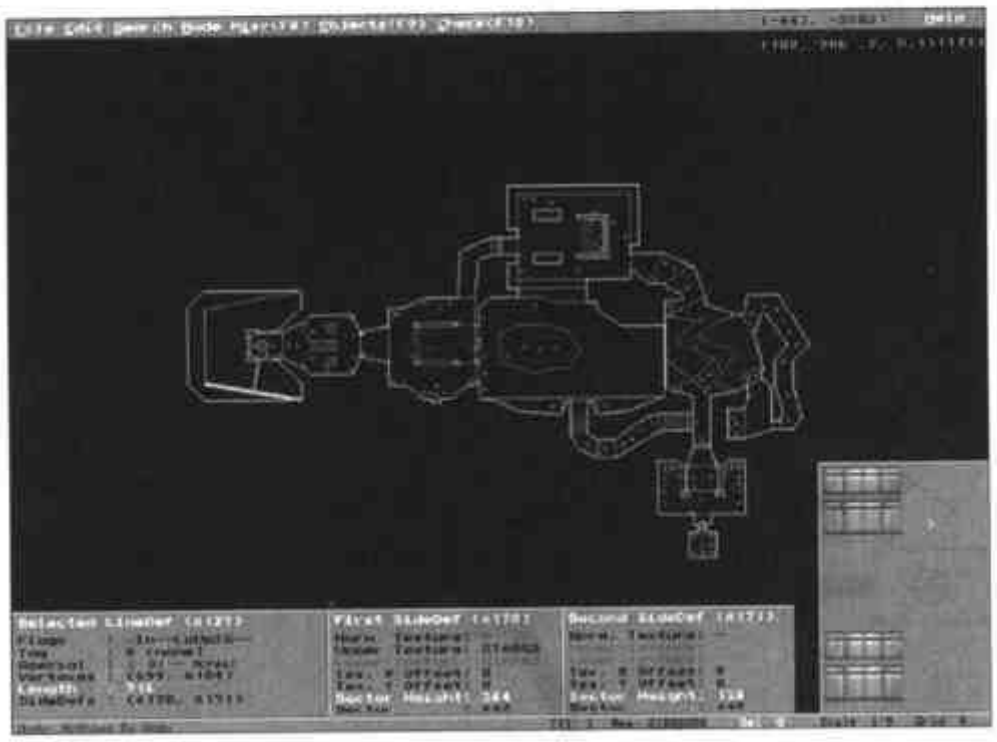


图 6.50　DoomEdit 的屏幕截图

最初，作者打算在本书中使用 Quake 格式的 3D 世界，并使用 WorldCraft 来创建 3D 世界，但 WorldCraft 的开发商陷入了困境，将这个程序转让给了 Valve 公司，现被更名为 Hammer Editor，而 Quake 是一种专用格式。考虑再三，作者不想受制于这种格式，决定使用诸如 Caligari .COB、3D Studio Max .3DS 或.MAX、.DXF、.NFF（neutral file format），让读者能够使用常见的工具和有大量文档（well-documented）的格式。另外，本书要介绍的是 3D 图形学和游戏编程，而不是要编写关卡设计工具，因此不打算偏离主题。因此，读者可以使用任何 3D 建模工具，只要它支持本书的引擎能够读取的文件格式。

基于这个原因，本书附带光盘中包含大量 3D 建模演示程序和一些非常不错的共享软件建模程序，如 Moray。即使这些共享软件建模程序不支持您想要的文件格式，也可以使用本书附带光盘中的 3D 文件格式转换器进行转换。

动画数据和运动数据

有时候，您可能需要创建关节型动物动画，这非常复杂。本书前面介绍过，可以使用网格动画或骨架动画来实现动物动画。采用第一种方法时，需要使用 3D 建模程序生成角色网格关键帧，然后读取这些关键帧。采用第二种方法时，需要读取包含关节点的网格，然后，通过网格链杆输出运动数据（如 BioVision 运动文件）以制作动物或物体动画。这同样是一项极其复杂的任务，工具能够给您提供物体甚至运动数据（或者自己编写或购买），但必须编写代码，将这些数据提供给引擎。这将在后面更详细地介绍。诸如 3D Studio Max、Maya 和 trueSpace 等工具都是不错的角色动画软件包，让您能够完成上述两项任务。然而，最终我们将使用一种更基本的关键帧系统，如 Quake 的.MD2 格式。

6.6　从外部加载数据

前面说过，必要时将编写代码，但旨在让读者领会问题的本质。这里介绍几种可以使用的文件格式，让读者知道它们是什么样的。对于文件格式的总体要求是，能够使用引擎所需的信息表示 3D 物体。现在，我们并不需要有关纹理、光照等方面的信息，而只需要描述物体的顶点列表和多边形网格。

然而，随着读者往下阅读，将发现我们需要知道很多信息。例如，如果使用 3D 建模程序创建坦克，将需要知道顶点列表、多边形网格、纹理坐标、多边形属性等，因此需要一种更复杂的文件格式。另外，如果使用 3D 建模程序创建游戏世界，建模程序必须支持引擎能够使用的导出光源和标记信息。

这里不打算详细描述各种文件格式，而只简要地描述一些比较流行的文件格式。

6.6.1　PLG 文件

PLG 文件格式是 REND386 设计人员发明的，其中的成员之一是 Bernie Roehl（实际上，作者认为所有工作都是由他完成的）。这种格式很简单，并抓住了要点，从它开始介绍是种不错的选择。不幸的是，它不支持纹理映射和双面多边形，但对于快速手工编辑模型而言足够了，且很灵巧。使用这种文件格式的主要原因是，有些 3D 建模程序能够生成 PLG 文件，同时有现成工具能够将.DXF 文件转换为 PLG 文件，更为重要的是，编写读取这种文件的分析程序只需要一两个小时，而对于更高级的文件格式，需要一周。

1. PLG 文件的首行

PLG 文件以物体的名称及其包含的顶点数和多边形数打头：

```
object_name num_vertices num_polygons
```

例如：

```
pyramid 5 5
```

表示接下来是物体 pyramid 的数据，这个物体有 5 个顶点和 5 个面。

2. PLG 顶点列表

接下来的数据是顶点，顶点按 0 到 n 的顺序排列。每个顶点的坐标之间用空格隔开，坐标排列顺序为 x-y-z。例如，对于如图 6.51 所示的棱锥，顶点数据如下：

```
0    20   0
 30  -10  30
 30  -10 -30
-30  -10 -30
-30  -10  30
```

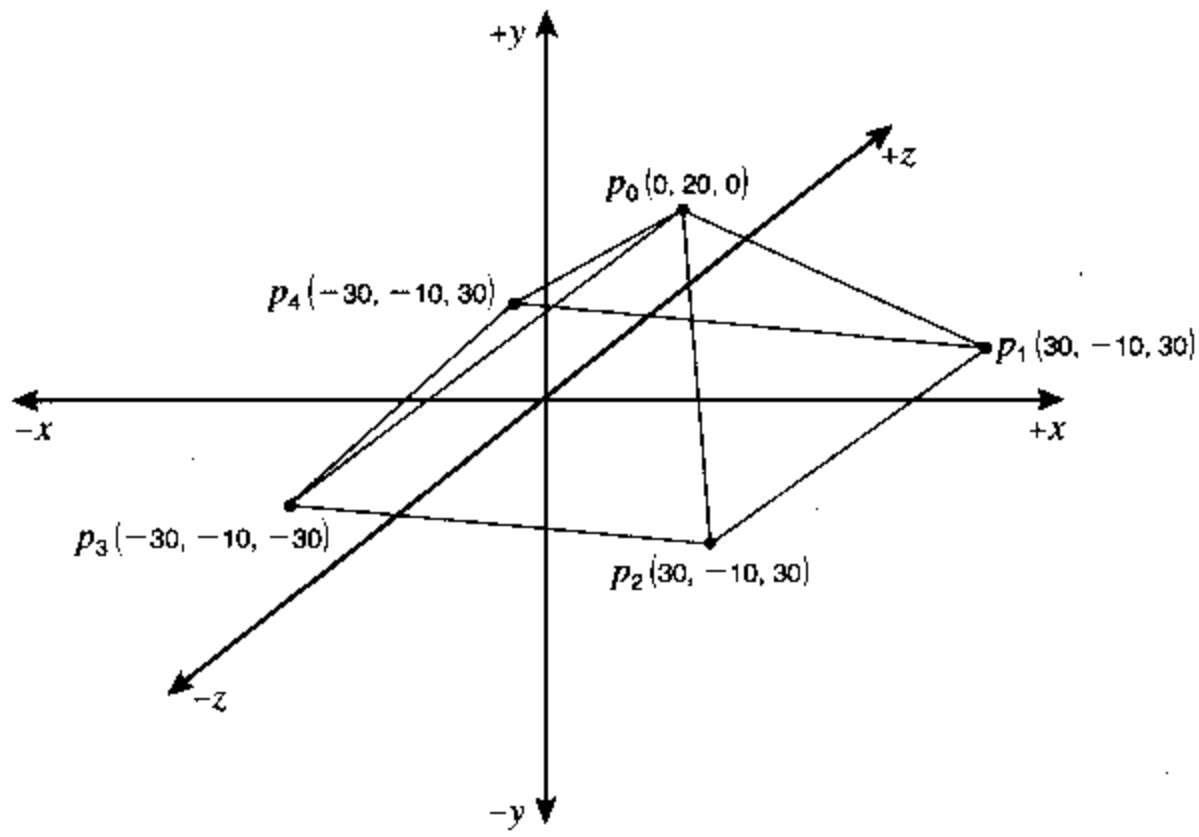


图 6.51　包含 5 个面的棱锥

3. PLG 多边形列表

文件的下一部分是多边形描述，每行描述一个多边形的颜色、顶点数、顶点的索引号，格式如下：

```
surface_description n  v1 v2 v3 ... vn
```

其中 surface_description 是面描述符（surface descriptor），稍后将介绍；n 是顶点数；v1、v2...vn 是顶

点，按反时针方向排列。对于前述棱锥，多边形列表如下：

```
10  3 0 2 1
10  3 0 3 2
10  3 0 4 3
10  3 0 1 4
10  4 1 2 3 4
```

这意味着有 5 个多边形面。每行中的第一个数字是面描述符，接下来是多边形的顶点数。例如，第二行中的多边形有 5 个顶点：0、3 和 2，第 5 五行的多边形有 4 个顶点：1、2、3 和 4。我们要求模型必须由三角形组成，这里包含四边形只是为了说明这种文件格式支持 n 边多边形。

上述数据中，惟一不那么容易理解的是每行中的第一个数字，它被称为面描述符（对于棱锥，所有面的面描述符都是 10）。下面介绍面描述符的含义。

4. PLG 面描述符

在每个多边形的定义中，第一个数字被称为面描述符，它可以是十进制整数或十六进制整数——0x 或 0X 后面跟一个十六进制整数值。面描述符是一个 16 位的值，根据各位的值按如下方式进行解释：

H_{15} R_{14} $S_{13}S_{12}$ $C_{11}C_{10}C_9C_8$ $B_7B_6B_5B_4B_3B_2B_1B_0$

- R 位被预留，应将其设置为 0；
- 如果 H 位为 1，表示面是带纹理的，最后 14 位为纹理图索引；
- 如果 H 位为 0，则按表 6.1 来解释 SS 位。

表 6.1　　PLG 面描述符中 SS 位的含义

$S_{13}S_{12}$ 的值	含义
00	多边形采用固定着色处理，应使用一种固定颜色绘制。如果 CCCC 为零，则 BBBBBBBB 指定了 8 位颜色模式下的 256 种颜色之一；如果 CCCC 不为零，则它们指定了 16 种色调（hue）之一，BBBBBBBB 的前 4 位指定了使用该色调的哪种着色度（shade）
01	多边形采用恒定着色处理，即绘制时根据光线照射到多边形的角度来确定颜色的亮度，因此当多边形移动时，其亮度将发生变化。CCCC 指定了 16 种色调之一，BBBBBBBB 表示颜色的亮度。该亮度值将被乘以多边形法线向量与多边形到光源的向量之间的交角的余弦，该乘积被用作索引，从指定颜色的着色度数组中找出相应的着色度。如果 CCCC 为 0，颜色总是为黑色
10	多边形为使用金属效果（metallic）。CCCC（不能为零）指定了 16 中色调之一，BBBBBBBB 的前 5 位指定了一个偏移量，将从这里开始在一个着色度范围内循环变化，以实现金属效果
11	应将多边形视为透明的（transparent）。和面类型 10 一样，但绘制多边形时不是使用固定的颜色，而是使用交替开/关的点图案，让您能够“看透”多边形

表 6.1 反映的是 PLG 文件格式的原始定义，它显然是为 4/8 位颜色模式设计的，因此显得有些过时。虽然，8 位颜色模式有时候是可行的，但我们确实需要使用 16 位颜色模式。因此，下一章使用这种文件格式时，将对其做细微的修改，以便能够支持 16 位 RGB 颜色。当然，导出文件的程序并不知道这一点，但我们至少能够为试验提供手工支持。

5. 加载 PLG 文件中的物体

加载 PLG 文件不太难。从本质上说，PLG 是标准 ASCII 文件，包含一行或多行描述物体的文本。惟一使得读取 PLG 文件有些复杂的因素是，面描述符可能是十进制的，也可能是十六进制的，这取决于是否有前导标记 0x，另外，文件中可以包含以字符#打头的注释。真正编写分析程序时，您将发现并没有想象的那么难；只不过是每次读取一行，并对其中的数字进行分析而已。相对来说，对于手工生成模型而言，这是一种不错的格式，下一章将实际使用这种格式。下面是物体的完整数据：

```
pyramid 5 5

# 顶点列表
```

```
 0    20   0
 30  -10   30
 30  -10  -30
-30  -10  -30
-30  -10   30

# 多边形列表

10  3 0 2 1
10  3 0 3 2
10  3 0 4 3
10  3 0 1 4
10  4 1 2 3 4
```

至此，对 PLG 格式的基本描述就结束了，读者应该明白作者为何喜欢这种格式。它简单、易懂、可作为学习的起点，同时编写读取这种格式的阅读器也很容易。当然，大多数建模程序都不直接支持这种格式，因为它太粗糙，但可以使用转换器进行转换。PLG 文件存在的惟一问题是，它采用右手坐标系统，顶点按反时针排列，因此加载从工具中导出的物体时，必须考虑到这一点，因为我们使用的是左手坐标系。

6.6.2　NFF 文件

下一种使用起来非常容易的文件格式是中立文件格式（neutral file format，NFF）。多年前，作者编写 VR 游戏时首次使用了这种格式，看起来很不错，因此决定支持它。这种格式最初是由 Sense-8 的 Eric Haines 开发的，那是很久以前了。NFF 文件格式的最新版本为 3.1，这里将介绍这种版本。

NFF 格式用于定义简单世界和物体，而 PLG 格式只能定义物体。NFF 格式支持诸如相机、背景色、光源位置等实体。这些特性对 NFF 格式的分析复杂度影响不大，因此对于我们要做的工作以及测试算法和试验而言，这种格式很不错。

1. NFF 支持的 3D 实体

NFF 支持下述实体（entity）：

- 简单视景体；
- 背景色描述；
- 定位（和定向）光源描述；
- 面属性描述；
- 多边形、多边形曲面片（polygonal patch）、圆柱/圆锥和球体描述。

NFF 文件由多行文本组成。每个实体占据一行或多行，第一个字段定义了其类型，其他内容定义了实体本身的其他信息。实体类型包括：

- v——观察向量和角度；
- b——背景色；
- l——定位光源；
- f——物体的材质属性；
- c——圆锥/圆柱图元；
- s——球体图元；
- p——多边形图元；
- pp——多边形曲面片图元。

接下来简要地描述一下我们将处理的一些较重要的实体。关键字将用粗体显示，参数用常规字体显示，后者通常是浮点值。

（1）视点位置格式

视点位置定义了视点以及前面介绍过的部分视景体参数，其格式如下：

```
v
```

```
from Fx Fy Fz
at Ax Ay Az
up Ux Uy Uz
angle angle
hither hither
resolution xres yres
```

上述参数的含义如下：

- from——用世界坐标表示的视点位置；
- at——将成为图像中心的“注视（look at）”位置，用世界坐标表示；
- up——一个向量，定义了一个相对于注视向量来说为上方的方向；
- angle——视野角度，单位为度；指的是从视点到最上面和最下面的像素行中点的向量的夹角以及从视点到最左边和最右边的像素列中点的向量的夹角；
- hither——近裁剪面到视点的距离；
- resolution——视平面的分辨率，单位为像素。

注意：这里没有对数据归一化做任何假设（例如，从视点到注视点的距离可以不为 1）；向量不必相互垂直；另外，远裁剪面默认位于无穷远处，高宽比默认为 1.0。

根据惯例，定义任何物体之前，必须定义观察实体，这样引擎将知道从哪里进行观察。

（2）背景色格式

背景色是用 RGB 定义的，其中每个分量的取值范围为 0～1：

```
b R G B
```

如果没有设置背景色，则认为其 RGB 值为{0, 0, 0}。

（3）定位光源格式

这种光源使用位置和颜色定义的：

```
l X Y Z [R G B]
```

其中 *X*、*Y*、*Z* 是光源的位置，R、G、B 是颜色（这是可选的）。同样，光源实体也必须在任何物体之前定义。如果没有设置光源的颜色，将默认使用一个非零强度，该强度未明确指定。

（4）填充颜色和着色处理参数格式

填充颜色和着色处理参数指定了接下来所有实体的光照属性，直到重新指定填充颜色和着色处理属性为止。

```
f red green blue Kd Ks Shine T index_of_refraction
```

上述各个参数的含义如下：

- red、green、blue——R、G、B 值，取值范围为 0.0～1.0；
- Kd——散射光反射分量；
- Ks——镜面反射光反射分量；
- Shinc——用于计算反射区域的 Phong 余弦指数；
- T——透射率。

由于我们还没有介绍光照计算，上述参数还并不重要；然而，一般而言，计算表面反射光中散射项和镜面反射项的公式中，使用了上述大部分参数。

（5）多边形格式

多边形是由一组顶点定义的，这些顶点必须共面。在 NFF 文件中，所有多边形都是单面的，在右手坐标系中，当观察者位于多边形前面时，顶点按逆时针方向排列。另外，前两条边必须构成一个非零凸角，以便只根据前 3 个顶点便可以确定法线和面可见性。

警告：NFF 文件格式是基于右手坐标系的。

```
p num_vertices
v1.x v1.y v1.z
v2.x v2.y v2.z
```

```
.
.
vn.x vn.y vn.z
```

其中 num_vertices 是多边形包含的顶点数，v 表示顶点。

（6）多边形曲面片的格式

多边形曲面片是由一组顶点及其法线定义的，除了每个顶点都有法线外，它几乎与多边形相同。

```
pp num_vertices
v1.x v1.y v1.z n1.x n1.y n1.z
v2.x v2.y v2.z n2.x n2.y n2.z
.
.
vn.x vn.y vn.z nn.x nn.y nn.z
```

其中 num_vertices 为多边形曲面片包含的顶点数，v 和 n 分别表示顶点及其法线。

注意：NFF 文件格式还有很多其他功能，如允许定义诸如球体和锥体等图元，但从很大程度上说，除注释字符#外，我们需要的所有实体都在前面介绍过。

2. NFF 文件示例

结束对 NFF 文件的简介之前，下面列出一个表示 2×2×2 立方体的 NFF 文件：

```
v
from 0 2.0 2.0
at 0 0 0
up 0 1 0
angle 40
hither 1
resolution 512 512
l 0 5 0
l 0 2.0 2.0
f 0.000000 0.000000 0.000000 1 0 0 0 0
p 3
1.000000 -1.000000 -1.000000
-1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
p 3
1.000000 -1.000000 -1.000000
-1.000000 -1.000000 1.000000
1.000000 -1.000000 1.000000
p 3
-1.000000 1.000000 -1.000000
-1.000000 -1.000000 -1.000000
1.000000 -1.000000 -1.000000
p 3
-1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 1.000000 -1.000000
p 3
1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 -1.000000 1.000000
p 3
1.000000 1.000000 -1.000000
1.000000 -1.000000 1.000000
1.000000 1.000000 1.000000
p 3
1.000000 1.000000 1.000000
1.000000 -1.000000 1.000000
-1.000000 -1.000000 1.000000
p 3
1.000000 1.000000 1.000000
-1.000000 -1.000000 1.000000
-1.000000 1.000000 1.000000
p 3
```

```
-1.000000 1.000000 -1.000000
1.000000 -1.000000 -1.000000
1.000000 1.000000 1.000000
p 3
-1.000000 1.000000 -1.000000
1.000000 1.000000 1.000000
-1.000000 1.000000 1.000000
p 3
-1.000000 -1.000000 -1.000000
-1.000000 1.000000 -1.000000
-1.000000 1.000000 1.000000
p 3
-1.000000 -1.000000 -1.000000
-1.000000 1.000000 1.000000
-1.000000 -1.000000 1.000000
```

6.6.3 3D Studio 文件

3D Studio 及其继任 3D Studio Max 是游戏和电影行业中使用最广泛的 3D 建模程序之一。最初的 3D Studio 面世了很长时间，它实际上是更古老的程序 AutoCad 的继任，后者是 AutoDesk 开发的。最先支持 DXF 文件格式的是 AutoCad，随着 3D Studio 的面世，3DS 文件格式和新的 MAX 文件格式也包含整个场景和每个网格的数据。不幸的是，这些文件格式非常复杂，可以是 ASCII 版本或二进制版本。由于篇幅有限，这里不打算像 PLG 和 NFF 那样详细介绍它们，而只对每种文件格式做简要的描述，并提供一个小型范例。然而，本书后面可能会支持 DXF 和/或 3DS 文件格式。

1. DXF 文件

DXF（Data eXchange Format，数据交换格式）最初是 AutoDesk 的 AutoCAD 支持的一种文件格式。这种格式有二进制版本和 ASCII 版本，但名称相同。这种格式还是不错的，但它并非为高端 3D 实体建模、光照处理和纹理映射等设计的，这是因为最初的设计人员更感兴趣的是建立 3D 机械和建筑模型，而不是从美学上说令人满意的图像。因此，这种文件格式提供了 3D 支持，但并不全面。另外，对于其支持的实体，DXF 文件中的开销非常大，同时技术图样中必须包含大量与物体本身无关的技术细节。因此，读取模型数据之前，通常必须分析所有无用的信息；但到达 3D 模型数据的位置后，读取文件将非常容易。

DXF 文件的格式

标准的 ASCII DXF 文件包含下列几个主要部分：

- HEADER——包含有关模型的通用信息；
- TABLE——包含指定项的定义；
- BLOCKS——包括块定义实体，它们描述了构成图样中每个块的实体；
- ENTITIES——包含描述模型本身的实体和块引用（block reference）；
- END OF FILE 标记——每个 DXF 文件都以字符串 EOF 结尾。

当然，这些部分中有各种各样的数据。数据被归类，并用组编码指示。组编码用于帮助编写分析函数，因为只要查看组编码，便可以知道数据的类型。表 6.2 列出了基本的组编码。

表 6.2　　DXF 文件中的组编码

组编码范围	含义	组编码范围	含义
0～9	字符串值	60～79	整数值
10～59、210～239	浮点值	999	注释

在 DXF 文件中，任何一项数据之前都有一个组编码，指出了数据的类型。因此，DXF 文件中的任何一个数字或字符串的前面都有一个组编码，这是很大的开销和浪费，但使得分析文件更容易。每个组编码范围都被分成多个子类，它们更详细地说明了数据，但 99%的组编码都与我们不相干。表 6.3 更详细地描述了组编码。

表 6.3　　DXF 文件中组编码的详情

组编码	含义	组编码	含义
0	标识文件、实体或表的开头	30	主 z 坐标
1	实体的主文本值（primary text value）	31～37	附加 z 坐标
2	某种形式的名称	38	高度（elevation）
3～4	其他文本或信息	39	厚度
6	直线类型	40～48	常规浮点值
7	直线样式	49	重复值
8	层名	50～58	角度
9	变量名标识符，用于 HEADER 部分中	62	色数
10	主 x 坐标	66	“实体跟踪（entities follow）”标记
11～18	附加 x 坐标	70～78	整数值
20	主 y 坐标	210、220、230	延展方向（extrusion follow）的 X、Y、Z
21～28	附加 y 坐标	999	注释

提示：有关 DXF 文件就介绍这么多，因为作者在 10 年前学习这种文件格式时，对各种数字感到极其迷惑，因此不希望读者有同感。不过，现在读者至少对各种数字的含义及为何使用它们有所了解。与 3D 数据相比，这些开销看起来大了些。

组编码总是在数据之前。例如，在 DXF 文件的开头，有一个字符串值标识文件的开始。从表 6.3 和表 6.4 可知，字符串值的组编码为 0～10，文件开头的组编码为 0，因此 DXF 文件的开头与下面类似：

```
0
SECTION
```

要加入注释，可以使用组编码 999，然后再加入注释，如下所示：

```
999
This is a comment, stupid huh!
```

下面来看一个普通的空 DXF 文件。其中使用 C++注释符“//”加入了注释，用于说明，DXF 文件中并没有这样的内容：

```
0 // 表示 HEADER 部分开头的组编码
SECTION

2 // 组编码
HEADER
 // 这里为 HEADER 项

0 // 组编码
ENDSEC   // 标识 HEADER 部分结束

0        // 表示 TABLES 部分开头的组编码
SECTION

2        // TABLES 组编码
TABLES

0        // TABLE 组编码
TABLE

2        // 视口 ENTITY 组编码
VPORT
70

// 视口表中最多包含多少项
```

```
// 视口表项

0        // 表示 TABLE 结尾的组编码
ENDTAB

0        // TABLE 组编码
TABLE

2        // 表示接下来为 LTYPE、LAYER、STYLE、VIEW、UCS 或 DWGMGR 的组编码
70
// 最大表项数
// 这里为表项

0        // 表示 TABLE 结尾的组编码
ENDTAB

0
ENDSEC    // TABLES 部分结尾标记

0     // 接下来为 BLOCKS 部分
SECTION

2     // BLOCKS 组编码
BLOCKS

// 这里为块定义实体

0
ENDSEC  // BLOCKS 部分结束标记

0        // 接下来为 ENTITIES 部分
SECTION

2       // ENTITIES 的开始位置
ENTITIES
// 这里为图样实体
0        // SECTION 部分结束组编码
ENDSEC // ENTITIES 结束标记

0
EOF // 文件末尾
```

正如读者看到的，即使是空的 DXF 文件，也包含不少的内容；但幸运的是，我们只关心 ENTITIES 部分，因为 3D 数据存储在这里。实体有很多，如表 6.4 所示。

表 6.4　　很有用的 DXF 文件实体

实体标签	描述	实体标签	描述	实体标签	描述	实体标签	描述
LINE	线段	ARC	弧	TEXT	文本字符串	VERTEX	顶点
POINT	点	TRACE	空间中的轨迹	SHAPE	形状	3DFACE	3D 面或多边形
CIRCLE	圆	SOLID	实心物体	POLYLINE	折线		

我们只关心实体 3DFACE，导出到 DXF 文件中时，任何 3D 几何体是以 3DFACE 的方式存储的。DFX 文件中没有光照、纹理映射和相机等信息，只包含非常简单的网格数据，仅此而已。四边形对应的 3DFACE 实体如下（这里也使用“//”来加入注释，但这是非法的；注释的组编码为 999）：

```
0 // 这是起始位置，因此使用组编码 0
3DFACE
10 // 接下来为顶点 0 的 x 坐标
x0

20 // 接下来为顶点 0 的 y 坐标
y0
```

```
30  // 接下来为顶点 0 的 z 坐标
z0

// 接下来为下一个顶点的坐标，因此将 X、Y、Z 的组编码都加 1

11  // 接下来为顶点 1 的 X 坐标
x1

21  // 接下来为顶点 1 的 y 坐标
y1

31  // 接下来为顶点 1 的 z 坐标
z1

// 接下来为下一个顶点的坐标，因此再次将 X、Y、Z 的组编码都加 1

12  // 接下来为顶点 2 的 x 坐标
x2

22  // 接下来为顶点 2 的 y 坐标
y2

33  // 接下来为顶点 2 的 z 坐标
z2

// 接下来为下一个顶点的坐标，因此再次将 X、Y、Z 的组编码都加 1

13  // 接下来为顶点 3 的 x 坐标
x3

23  // 接下来为顶点 3 的 y 坐标
y3

33  // 接下来为顶点 3 的 z 坐标
z3
```

注意：要定义三角形，只需将最后一个顶点 v3 的值设置成与 v2 相同。

下面是立方体的 DXF 文件（在 DXF 文件中，数据是逐行排列的，这里将每行包含多列，以便读者能够一眼看到文件的大部分内容）：

```
0               0               0               0
SECTION         SECTION         3DFACE          3DFACE
2               2               8               8
HEADER          HEADER          CUBE            CUBE
0               0               10              10
ENDSEC          ENDSEC          -1.000000       1.000000
0               0               20              20
SECTION         SECTION         1.000000        1.000000
2               2               30              30
TABLES          TABLES          0.000000        2.000000
0               0               11              11
TABLE           TABLE           -1.000000       1.000000
2               2               21              21
LAYER           LAYER           -1.000000       -1.000000
70              70              31              31
153             153             0.000000        2.000000
0               0               12              12
LAYER           LAYER           1.000000        -1.000000
2               2               22              22
Cube            Cube            -1.000000       -1.000000
70              70              32              32
0               0               0.000000        2.000000
62              62              13              13
15              15              1.000000        -1.000000
6               6               23              23
CONTINUOUS      CONTINUOUS      1.000000        1.000000
0               0               33              33
ENDTAB          ENDTAB          0.000000        2.000000
```

```
0               0               62              62
ENDSEC          ENDSEC          0               0
0               0               0               0
SECTION         SECTION         3DFACE          3DFACE
2               2               8               8
ENTITIES        ENTITIES        CUBE            CUBE
0               0               10              10
3DFACE          3DFACE          1.000000        -1.000000
8               8               20              20
CUBE            CUBE            1.000000        1.000000
10              10              30              30
1.000000        1.000000        0.000000        0.000000
20              20              11              11
-1.000000       -1.000000       1.000000        1.000000
30              30              21              21
0.000000        0.000000        -1.000000       1.000000
11              11              31              31
-1.000000       -1.000000       0.000000        0.000000
21              21              12              12
-1.000000       -1.000000       1.000000        1.000000
31              31              22              22
0.000000        0.000000        -1.000000       1.000000
12              12              32              32
-1.000000       -1.000000       2.000000        2.000000
22              22              13              13
-1.000000       -1.000000       1.000000        -1.000000
32              32              23              23
2.000000        2.000000        1.000000        1.000000
13              13              33              33
1.000000        1.000000        2.000000        2.000000
23              23              62              62
-1.000000       -1.000000       0               0
33              33
2.000000        2.000000
62              62
0               0
```

注意：在上述内容中，每个 3DFACE 的数据之前都有 8 和 CUBE。这是完全合法的，也是为何使用组编码的原因所在。在这里，8 是层名的组编码，字符串 CUBE 是层名，它是被导出的 3D 物体的名称。

正如读者看到的，分析这个文件没有想象的那么难。只需找到 ENTITIES 部分，然后开始读取每个 3DFACE 即可。惟一令人不快的是，DXF 文件没有使用顶点列表，而是使用独立的顶点定义了每个面。

2. 3DS/ASCII 文件

3D Studio Max 支持很多文件格式，但其中最天然（native）的是 3DS 和 ASCII。它们实际上是同一种格式，但 3DS 是二进制的，而 ASCII 是人类能够读懂的。这些格式极其强大，可使用它们来完成任何工作。这里不打算描述它们，而只列出位于(0，0，0)的 2×2×2 立方体的文件内容，通过阅读该文件，读者将能够大概了解这种格式：

```
Ambient light color: Red=0.3 Green=0.3 Blue=0.3

Named object: "Cube"
Tri-mesh, Vertices: 8     Faces: 12
Vertex list:
Vertex 0:  X:-1.000000    Y:-1.000000    Z:-1.000000
Vertex 1:  X:-1.000000    Y:-1.000000    Z:1.000000
Vertex 2:  X:1.000000     Y:-1.000000    Z:-1.000000
Vertex 3:  X:1.000000     Y:-1.000000    Z:1.000000
Vertex 4:  X:-1.000000    Y:1.000000     Z:-1.000000
Vertex 5:  X:1.000000     Y:1.000000     Z:-1.000000
Vertex 6:  X:1.000000     Y:1.000000     Z:1.000000
Vertex 7:  X:-1.000000    Y:1.000000     Z:1.000000
Face list:
Face 0:    A:2 B:3 C:1 AB:1 BC:1 CA:1
```

```
Material: "r255g255b255a0"
Smoothing:   1
Face 1:    A:2 B:1 C:0 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:   1
Face 2:    A:4 B:5 C:2 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:   1
Face 3:    A:4 B:2 C:0 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:   1
Face 4:    A:6 B:3 C:2 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:   1
Face 5:    A:6 B:2 C:5 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:   1
Face 6:    A:6 B:7 C:1 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:   1
Face 7:    A:6 B:1 C:3 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:   1
Face 8:    A:6 B:5 C:4 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:   1
Face 9:    A:6 B:4 C:7 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:   1
Face 10:    A:1 B:7 C:4 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:   1
Face 11:    A:1 B:4 C:0 AB:1 BC:1 CA:1
Material:"r255g255b255a0"
Smoothing:   1
```

6.6.4　Caligari COB 文件

这些年来，作者已成为 Caligari trueSpace 拥趸。这是一个功能非常强大的建模程序，谁说它的界面不是世界上最容易使用的，我跟谁急。在包含的数据方面，COB 格式与 3DS 格式类似，这种格式也有二进制版本和 ASCII 版本。这种格式过于复杂，这里无法详细介绍；和前面一样，这里也列出立方体的 COB 文件，这比用几页篇幅进行介绍更有效：

```
Caligari V00.01ALH
PolH V0.06 Id 16796788 Parent 0 Size 00000981
Name Cube
center 0 0 f
x axis 1 0 0
y axis 0 1 0
z axis 0 0 1
Transform
1 0 0 0
0 1 0 0
0 0 1 1
0 0 0 1
World Vertices 8
-1.000000  -1.000000  -1.000000
-1.000000  -1.000000   1.000000
 1.000000  -1.000000  -1.000000
 1.000000  -1.000000   1.000000
-1.000000   1.000000  -1.000000
 1.000000   1.000000  -1.000000
 1.000000   1.000000   1.000000
-1.000000   1.000000   1.000000
Texture Vertices 14
0.000000 0.333333
```

```
0.000000 0.666667
0.250000 0.333333
0.250000 0.666667
0.500000 0.000000
0.500000 0.333333
0.500000 0.666667
0.500000 1.000000
0.250000 0.000000
0.250000 1.000000
0.750000 0.333333
0.750000 0.666667
1.000000 0.333333
1.000000 0.666667
Faces 6
Face verts 4 flags 0 mat 0
<2,2> <0,0> <1,1> <3,3>
Face verts 4 flags 0 mat 0
<4,4> <0,8> <2,2> <5,5>
Face verts 4 flags 0 mat 0
<5,5> <2,2> <3,3> <6,6>
Face verts 4 flags 0 mat 0
<6,6> <3,3> <1,9> <7,7>
Face verts 4 flags 0 mat 0
<4,10> <5,5> <6,6> <7,11>
Face verts 4 flags 0 mat 0
<0,12> <4,10> <7,11> <1,13>
DrawFlags 0
Radiosity Quality: 0
Unit V0.01 Id 16796789 Parent 16796788 Size 00000009
Units 2
Mat1 V0.06 Id 16799524 Parent 16796788 Size 00000090
mat# 0
shader: phong  facet: auto32
rgb 1,1,1
alpha 1  ka 0.23  ks 0.81  exp 0.91  ior 1
ShBx V0.00 Id 16799525 Parent 16799524 Size 00000627
Shader class: color
Shader name: "chrome" (chrome)
Number of parameters: 3
base colour: color (255, 255, 255)
vector: vector (0, 1, 0.5)
mix: float 0.5
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Shader class: reflectance
Shader name: "caligari phong" (caligari phong)
Number of parameters: 8
ambient factor: float 0.23
diffuse factor: float 0.19
specular factor: float 0.81
exponent: float 43.8135
specular colour: color (61, 67, 255)
transmission factor: float 0
mirror factor: float 0.78084
refraction: float 1
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
END  V1.00 Id 0 Parent 0 Size       0
```

正如读者看到的，与3DS格式相比，COB的开头增加了一些内容，但它基本上也是一种基于顶点的格式，包含一个清晰的面列表。另外，其内容几乎是通俗易懂的英语，读者只需阅读一个物体的这种文件，花两三个小时便能理解这种格式的大部分内容，如朝向、光源、纹理坐标等，而无需阅读有关这种格式的文档。

另外，trueSpace还有一种名为.SCN的格式，这种格式支持整个场景和世界。3D Studio Max也通过.MAX

格式支持这种功能。因此，使用这两种建模程序（Maya 或 Lightwave），不但可以定义问题，还可以定义包含光源等的世界。正如前面指出的，打算编写游戏时，应了解是否有现成的 3D 建模程序和自定义插件代码可用作游戏编辑器。大多数时候，这种方法的效果都不错，开发商正使 3D 建模程序使用起来日益简单，并在其新版本的工具中为游戏设计人员和编辑人员提供了越来越多的支持。

提示：3D Studio Max 有一种名为 MaxScript 的脚本语言。使用 MaxScript，几乎可以操纵 3D 世界中的任何东西。因此，可以通过编写脚本或宏，在 3D Studio Max 中加入新的功能，将其变成一个游戏编辑器。

6.6.5　Microsoft DirectX .X 文件

最后是 Microsoft 的 DirectX 格式。对于这种格式，作者不太在行，用得也不多，只是将其导出，然后让 Direct3D 为我导入过。也就是说，作者从来没有编写过这种文件的分析函数，但有一点可以告诉读者，这种格式的功能非常全面。它非常复杂，功能也非常强大，至于编写这种文件的分析函数，想都不要想。因此，本书将不编写处理.X 文件的程序。

6.6.6　3D 文件格式小结

正如读者看到的，每种 3D 文件格式都有其优缺点，但有 ASCII 版本的文件格式更佳。更多有关 3D 文件格式的信息，请阅读 Keith Rule 编写的 *3D Graphics File Formats*（Addison-Wesley Press）或在 Internet 上搜索。例如，下面两个 URL 是了解图形文件格式的好去处：

http://www.informatik.uni-frankfurt.de/~amueller/FF02.HTM

http://astronomy.swin.edu.au/~pbourke/3dformats/

对于本书需要的每种文件格式，我们都将编写一个分析函数。

注意：前面介绍的大部分文件格式都使用右手坐标系来导出坐标；然而，根据使用的工具，这种设置是可以改变的。作者想强调的是，编写分析函数时，必须考虑这一点；将物体加载到游戏中时，如果由于某种原因一切都是相反的，只需将顶点排列顺序反转并将 z 轴翻转即可。

6.7　基本刚性变换和动画

作为本书的读者，您应该牢固地掌握了 2D 动画技术，如平移、位图缩放和旋转。然而，在 3D 空间中，还可以用 3D 物体执行大量其他的操作，因为移动物体时有 6 个自由度，可以让网格本身变形。

本书前面概述了关键帧动画和基于运动数据的动画，本书后面介绍角色动画时将实现这几种动画。接下来介绍一些用于移动 3D 物体和改变其朝向的基本动画技术，这样您至少能够创建这样的游戏：其中有能够移动和做事情的刚性物体。下面简要地讨论如何实现基本的动画技术：运动、旋转和变形（morphing）。

6.7.1　3D 平移

3D 平移很简单，我们已经介绍过如何以手工方式和矩阵方式完成这种变换，这里不再赘述。这里要讨论的是如何平移整个网格，准确地说，是在 3D 游戏中平移网格的最佳方式。

每个物体都有一组相对于局部原点(0，0，0)的局部/模型坐标，但我们并不想通过平移局部坐标来平移物体。更好的方法是，平移物体的位置，然后在局部坐标到世界坐标变换中使用它作为参考点来平移物体，我们在前面这样做过。事实上，我们的物体数据结构中有一个记录物体位置的字段，它名为 world_pos。我们将平移网格中每个点的工作留给局部坐标到世界坐标变换去完成，这样只需处理平移单个点的工作，便可以移动整个物体。要根据向量 t = <tx，ty，tz>移动一个物体，只需这样做即可：

```
object.world_pos.x+=t.x;
object.world_pos.y+=t.y;
object.world_pos.z+=t.z;
```

非常简单。知道如何对单个点执行一次平移后，接下来的问题是如何在 3D 空间中移动物体？这实际

上是一个 AI 问题。AI 将生成一个合适的运动（平移值）流。但出于好玩，来尝试在 3D 空间中让物体绕一个圆移动。我们要做的是，计算绝对位置值或相对于前一个位置的位置值。出于简化的目的，我们使用绝对值而不是相对值。假设要让物体沿一个与 x-z 平面平行的椭圆轨迹移动，则代码如下：

```
float x_pos = a*cos(angle);
float y_pos = altitude;
float z_pos = b*sin(angle);
```

只需代入 a、b、angle 和 altitude 的值，上述数学公式便会输出椭圆轨迹上的位置。那么，绕螺旋线轨迹移动呢？这也很容易：

```
// 初始状态
float altitude = 0;
float rate     = .1;

// 每次循环都执行的代码
float x_pos = a*cos(angle);
float y_pos = altitude+=rate;
float z_pos = b*sin(angle);
```

这样，点(x_pos，y_pos，z_pos)将沿从地面($y = 0$)出发，沿螺旋线向上移动。正如读者看到的，编写 3D 运动代码和 2D 运动代码一样简单，即使沿曲线运动也没有任何问题。

6.7.2 3D 旋转

旋转比平移要棘手些。涉及的数学计算不难（虽然比平移复杂得多），难的是跟踪朝向。例如，将物体移到(x，y，z)处时，您知道它位于(x，y，z)处。但对旋转(ang_x，ang_y，ang_z)后，物体的朝向相对于坐标轴的角度并不一定是(ang_x，ang_y，ang_z)——还记得万向节死锁吗？另外，对物体进行旋转时，旋转顺序很重要；也就是说按 XYZ 和 YXZ 顺序旋转的结果完全不同。当然，如果旋转的是星星，可创建一个旋转速率(rx，ry，rz)，在每帧中据此对星星进行旋转，谁也不会注意到。但如果旋转的是坦克或飞船，则必要时必须计算它们的朝向。

在 3D 空间中旋转物体时，应对物体的本地坐标进行旋转。在局部坐标空间((0，0，0)为物体的局部坐标原点）而不是世界坐标空间进行旋转变换，否则物体旋转时将偏离中心。执行旋转并将旋转后的坐标存回到局部坐标数组中，还是在观察变换中考虑每个物体的当前朝向，完全取决于您。后一种方法指的是，当物体为渲染准备就绪后，对物体旋转一定的角度，并将结果存储到一个临时数组中。这种方法的优点是不会毁坏模型，可将其作为主拷贝，并将其他物体与之关联起来；其缺点是，对于不运动或非常稳定的物体，每帧中也必须经过旋转变换，从而浪费 CPU 周期。

有鉴于此，旋转物体时，应对其局部坐标进行变换，并将结果存储到局部坐标数组中。例如，假设在坦克模拟程序中，定义并加载了一个名为 tank1 的坦克物体，其朝向为+z 轴。当玩家将坦克向左或向右转的，我们可以旋转其局部坐标：

```
// 初始化
// 假设坦克在 x-z 平面中移动，因此旋转是绕 y 轴进行的
tank1.dir.x = 0;
tank1.dir.y = 0;
tank1.dir.z = 0;

// 游戏代码
if (turn_right)
   {
   for (int index=0; index<tank1.num_vertices; index++)
        {
        // 对每个顶点进行旋转，假设下述函数完成这种任务
        Rotate_Point_On_Y(&tank1.vlist_local[index], -5);
        // 更新当前朝向，以便知道坦克的朝向
        tank1.dir.y-=5;
        } // end for

   } // end if
else
```

```
if (turn_left)
    {
    for (int index=0; index<tank1.num_vertices; index++)
        {
        // 对每个顶点进行旋转，假设下述函数完成这种任务
        Rotate_Point_On_Y(&tank1.vlist_local[index], 5);

        // 更新当前朝向，以便知道坦克的朝向
        tank1.dir.y+=5;
        } // end for

    } // end if
```

上述代码很简单：如果玩家左转，则将整个顶点列表绕 y 轴旋转+5 度；如果玩家右转，则将顶点列表旋转-5 度。然而，旋转 3D 物体后，仅根据网格无法确定其朝向。

因此，通过更新 tank1.dir 来记录这种信息。在任何时候，根据 tank1.dir 都可以知道坦克朝向与+X 轴之间的夹角（绕 y 轴旋转时，我们假设+X 轴方向的角度为 0 度）。

这种方法很好，但同时绕 3 个轴旋转时，它便不再管用。前面说过，物体的实际朝向可能不是分别绕 3 个轴旋转的综合结果。然而，有一种更好的朝向记录方法——使用一个朝向向量。也就是说，对于 3D 世界中的每个物体，当您加载它并将其朝向设置为与+Z 轴平行时，初始化一个也与+Z 轴平行的单位向量。然而，每次对物体进行变换时，同时对该单位向量进行变换，如图 6.52 所示。这样，即使经过 1 000 000 次旋转后，该向量仍反映了物体的当前朝向。这就是字段 dir 的用途。然而，如果通过绕 Z-轴旋转，将物体倒转将如何呢？方向向量仍是正确的，但您丢失了信息，因为向量不能记录“倾侧角（roll）”（但四元数可以）。

重要的是，如果要使用向量来跟踪绝对朝向，需要在物体结构中使用 3 个向量，如图 6.53 所示。您首先将这些向量初始化为物体的局部坐标轴 x、y 和 z，然后在旋转物体时，对这 3 个方向向量也进行旋转。这样，您总是能够知道物体的绝对朝向，因为每当您对物体进行旋转时，对局部坐标轴也进行了相同的旋转，因此它们始终反映了所做的变换。

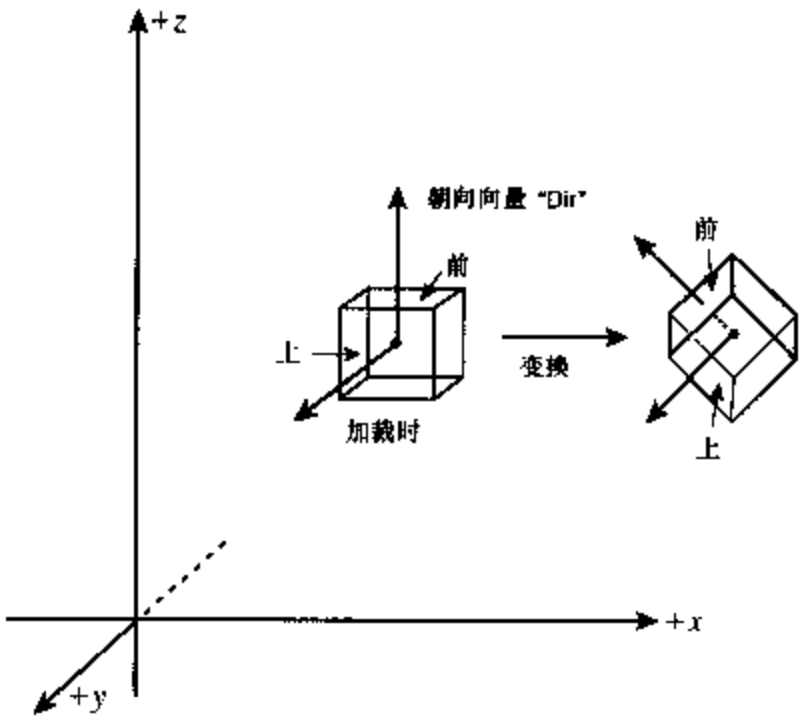


图 6.52　跟踪物体被加载和变换后的朝向

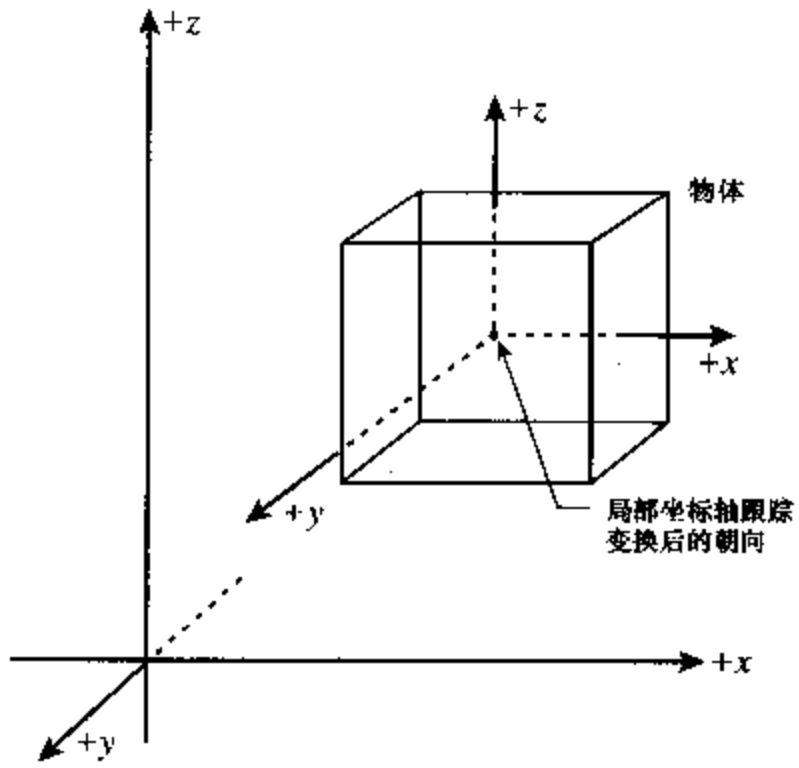


图 6.53　存储局部坐标轴以确定物体的绝对朝向

6.7.3　3D 变形

可对 3D 网格执行的最后一种简单动画是变形。这里指的不是将网格从一种形状变成另一种形状，而是实时地对网格数据进行缩放和剪切（shearing），使其看起来栩栩如生。例如，可以使用下面的代码来实现看似在呼吸的效果：

```
int up_down = 1; // -1 表示缩小，1 表示放大
```

```
float scale = 1; // 当前缩放比例

while(1)
{
// 缩小还是放大?
if (up_down==1)
    {
    Scale_Object(&object, 1.05);
    scale*=1.05;
} // end scale
else
    {
    Scale_Object(&object, .95);
    scale*=.95;
    } // end else

// 检测是否到达最大缩放程度
// 如果是，则反转缩放方式
if (scale < .75 || scale > 1.25)
    up_down=-up_down;

} // end while
```

上述代码每次将物体缩放 5%，使其看起来像在呼吸。当然，我们必须编写函数 Scale_Object()，该函数遍历传入物体的顶点列表，将每个顶点的坐标乘以缩放因子。作者想说的是，这里假定每次循环都对局部坐标进行修改，这很好，但数值误差将在模型中逐渐累积，根据模型数据，经过数百、数千或数万次修改后，模型将失真。对于旋转动画来说，也存在这样的问题。

警告：对局部坐标进行修改时，存在这样的可能性，即由于数值误差，经过多次修改后，最初的网格数据将丢失。

对于这种问题，惟一的解决方案是：要么保存每个网格的主拷贝并频繁地刷新每个物体；要么在流水线中动态地对物体的局部坐标执行旋转和变形操作，并将结果存储到一个临时数组中，而不是覆盖原始网格数据。

6.8 再看观察流水线

读者对 3D 流水线的各个部分有一定的了解后，下面更新流水线，在其中加入尽可能多的内容，并注出我们知道需要做大量工作的地方。

图 6.54 非常详细地说明了 3D 流水线，其中包含各个步骤。下面是主要的组成部分：

- 局部变换——对每个物体的局部坐标进行变换。可能在原地执行旋转、缩放或其他顶点操作（即修改局部坐标——译者注）。
- 局部坐标到世界坐标变换——根据物体的位置（它记录在 world_pos 中，是相对于物体的局部中心的）将局部坐标变换为世界坐标。
- 隐藏面消除和光照计算——消除 3D 模型中的隐藏面以及将整个物体从 3D 流水线中删除。另外，光照计算通常是在世界坐标空间中进行的。
- 物体消除——这个步骤有时候是单独进行的，可以与光照计算步骤分开。同时执行光照计算和隐藏面消除的惟一原因是，这样光照模型可以重用执行隐藏面消除时计算得到的面法线。
- 世界坐标到相机坐标变换——根据相机位置和视角对可能可见的几何体进行变换。
- 3D 裁剪——根据 3D 视景体对多边形进行裁剪。
- 透视投影和裁剪——将相机坐标变换为透视坐标。另外，透视变换可能将视景体归一化为单位立方体，使 3D 裁剪更容易。因此，有时候 3D 裁剪是在透视变换后进行的。
- 屏幕投影——这几乎是流水线的最后一站。在这里，将视平面坐标映射为屏幕坐标，并将其传递给光栅化函数进行最后的渲染。

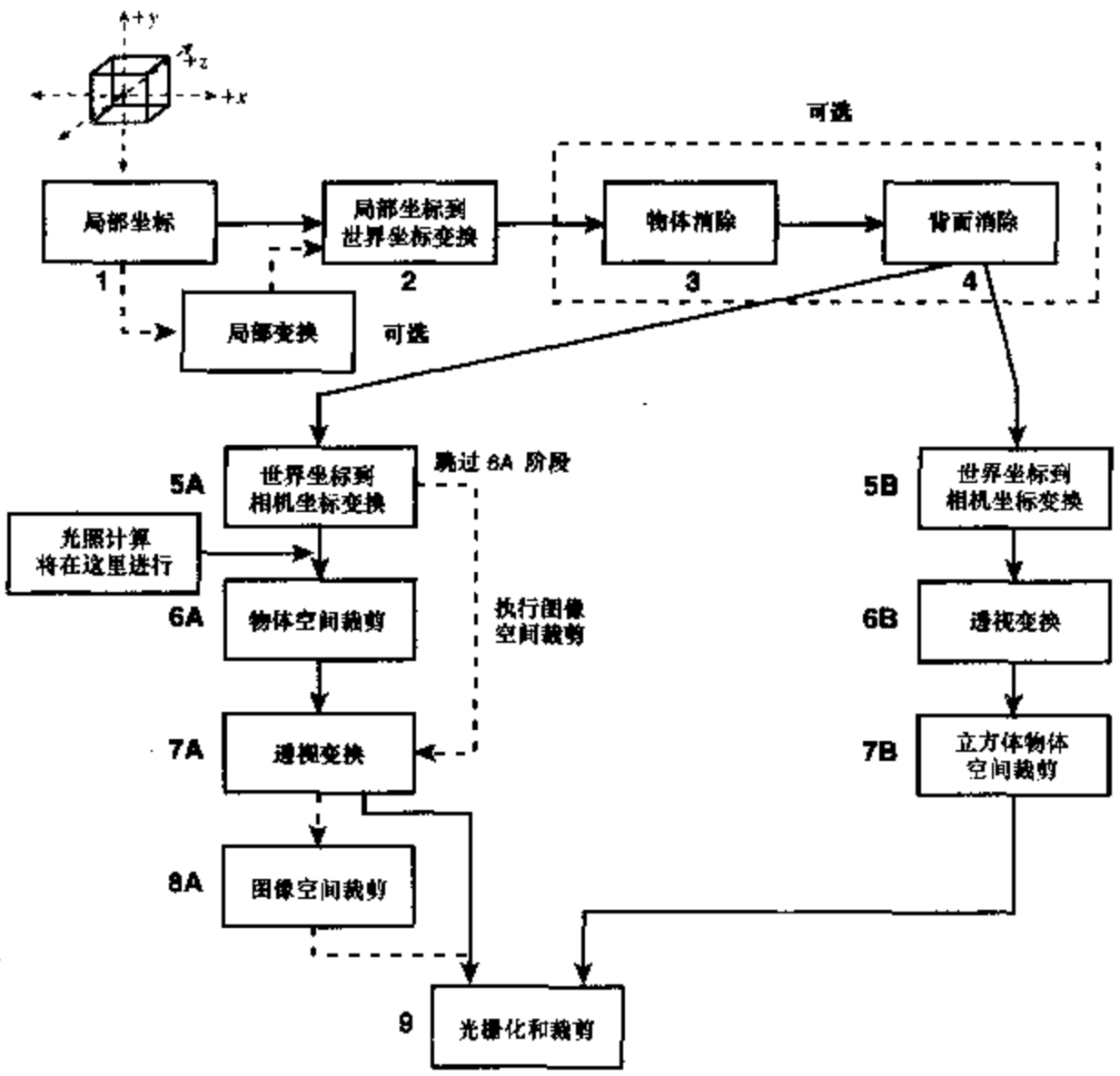


图 6.54　最后的 3D 流水线

● 光栅化和裁剪——这是 3D 流水线的最后一个阶段。在这里，将一个多边形列表（其中包含顶点颜色、纹理图等）传递给光栅化函数，后者将多边形渲染到屏幕上。另外，有些多边形可能穿过了屏幕边界（如果以前没有进行裁剪），将在像素级或扫描线级对它们进行裁剪。

必须承认，要使该流水线管用，还有很多细节需要解决；但从很大程度上说，流水线就是这样的。虽然对于光照处理我们介绍得不多，但对其涉及的内容读者可以想象得到：数学运算、光源和多边形的朝向，这些将在后续章节中讨论。

6.9　3D 引擎类型

接下来花些时间来介绍 3D 引擎类型。这里只想让读者对 3D 引擎有大概的了解——它们存在的原因和擅长做什么，而不过多地涉及其中的技术。

6.9.1　太空引擎

图 6.55 是优秀的老式太空游戏——Star War: X-Wing 的屏幕截图。正如读者看到的，该游戏大部分是由在空中移动的物体组成的。这种游戏编写起来比第一人称射击游戏容易得多，其原因很多，包括动画、碰撞检测和数据库问题等，然而，最基本的原因是其 3D 引擎不那么复杂。在大多数情况下，3D 太空游戏是基于物体的，这意味着所有的实体都是物体，在渲染之前，很多物体已经从流水线中删除；然后在渲染期间，使用简单的画家算法或 Z 缓存对组成每个物体的多边形进行排序。然而，由于物体表示的是太空中的飞船，因此很容易对物体进行排序。

另外，当前，大多数太空游戏都是使用多边形引擎编写的；然而，游戏一流的 3D 太空游戏（如 Wing Commander）是使用子画面引擎（sprite engine）编写的，使用了大量预先渲染好的风景（view）、快速位图倒影（bitmap reflection）和缩放算法。飞船等的位置都是用 3D 空间中的单个点表示的，因此从某种意义

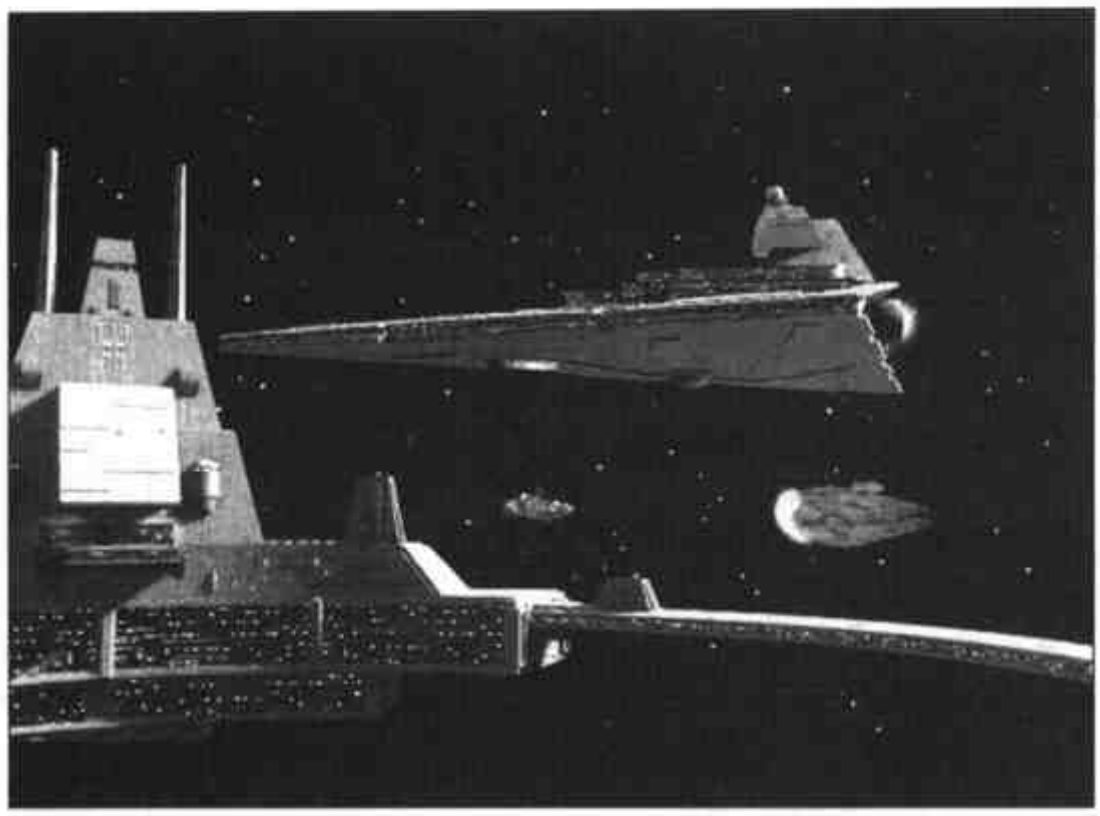


图 6.55　Star Wars: X-Wing 的屏幕截图

上说引擎是 3D 的，但模型不是。总之，当前的太空游戏在很大程度上说是使用多边形引擎编写的，光照、动画等都是以常规方式处理的。

在太空游戏中，比较难处理的是运动。在第一人称射击游戏中，模拟两足动物行走非常困难，但常常有预先录制好的动画文件，且通常是在 2D 表面（如地面）上行走。3D 太空游戏的问题与其说是渲染，不如说是太空飞船的运动和 AI。这更为复杂，因为飞船必须能够在 3D 空间中平稳地转向、翻滚和运动。这意味着可能必须使用 3D 曲线和其他技术。但一条经验法则是，为简化渲染算法和数据库工作方式，3D 游戏应为基于地面的（在沙漠或太空中）。

6.9.2　地形引擎

比太空引擎复杂点的是地形引擎。当然，地形引擎不仅仅是地形，但地形是游戏世界的一个主要组成部分，还必须有确定可见多边形的合适方法。图 6.56 是 Longbow Digital Arts 制作的游戏 Tread Marks 的屏幕截图，这是一个使用了大量地形的 3D 游戏。除地形外，地形游戏中的物体都很容易处理，它们可能在



图 6.56　最优秀的游戏 Tread Marks 的屏幕截图

地形上移动（飞行），在大多数情况下是坦克或车辆，因此动画制作不那么困难。当然，制作机械战士游戏时，必须处理复杂的动画和两足动物的地形跟踪问题。然而，地形引擎的主要问题是，如何表示世界数据库，它可能非常大。

例如，假设要创建一个大小为 100 000×100 000 单位的多边形网格世界，其中每个多边形的大小为 200×200 单位。这意味着该多边形网格包含大约 250 000（(100 000/200)*(100 000/200)）个多边形。

明白了作者的意思吧？需要跟踪大量的多边形。因此，必须使用某种空间划分方案，将任何时候可能可见的活动多边形数量降低到合理的程度。一种这样的技术是，将世界进行分区（sectorizing）。例如，如图 6.57 所示，100 000×100 000 的游戏世界被划分成 8×8 的分区阵列，每个分区包含大约 1000（250 000/256）个多边形，这是容易处理的。假设玩家的视野范围为 1000 个单位，则在最糟糕的情况下（玩家位于某个分区的角上），玩家也只能看到 4 个分区，这意味着在任何时候，最多只需处理 4000 个多边形。

同样，由于地形和大型数据库，基于地形或陆地的游戏比太空游戏复杂些，但不同的是，渲染可以使用改进的画家算法、二元空间划分（如果游戏世界非常大）或 z 缓存来完成。当然，地形引擎中另一个难以处理的问题是物体的移动以及如何使物体紧贴地形。后者被称为地形跟踪，指的是确保游戏物体的脚、轮胎等紧贴地面，同时其质心的向上向量垂直于当前的地形分片。

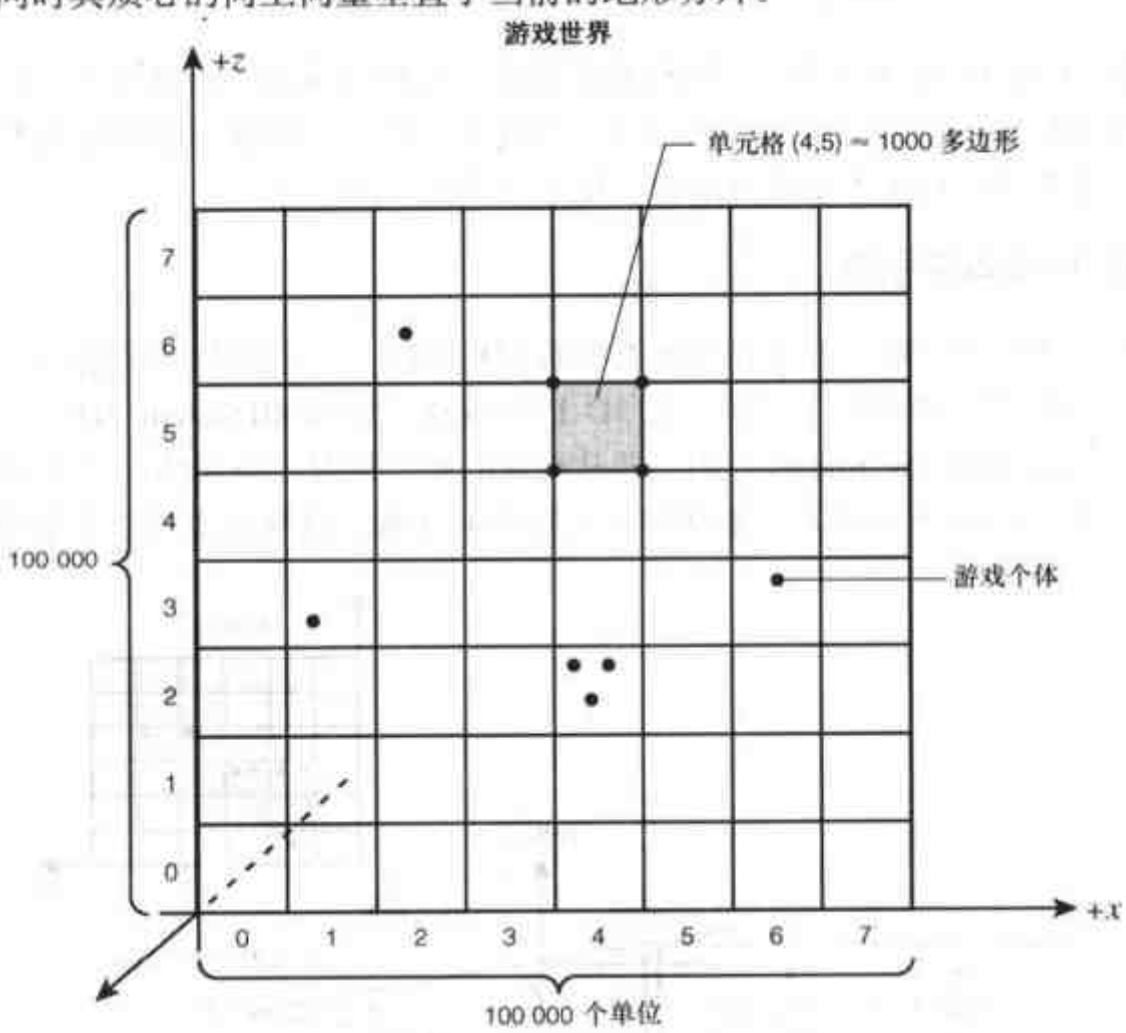


图 6.57　将大型世界划分成更小的分区

6.9.3　FPS 室内引擎

这种引擎比较棘手。图 6.58 是 Unreal Tournament 的屏幕截图，它是市面上最优秀的 3D 第一人称射击游戏（FPS）之一。第一人称射击游戏是最难编写的游戏，其原因有很多。首先，玩家在大部分时间都在室内，这意味着清晰的细节和近距离；其次，FPS 游戏世界在日益增大，多边形数据库也将随之增大。这意味着不能简单地将整个世界传递给 3D 流水线，而必须使用空间划分技术将游戏世界分区，以最大限度地减少需要考虑的多边形。

这意味着需要使用八叉树、二元空间划分、入口引擎等。在太空游戏或室外地形游戏中，3D 模型看起来不正确、多边形排序不正确等无关紧要；但在 FPS 游戏中，这样绝对行不通。请看图 6.58，FPS 游戏坚如磐石，让玩家沉醉在游戏世界中。玩家绝对不能接受不稳定、有缺陷或丑陋的游戏世界的，它们必须看起来像真的一样。

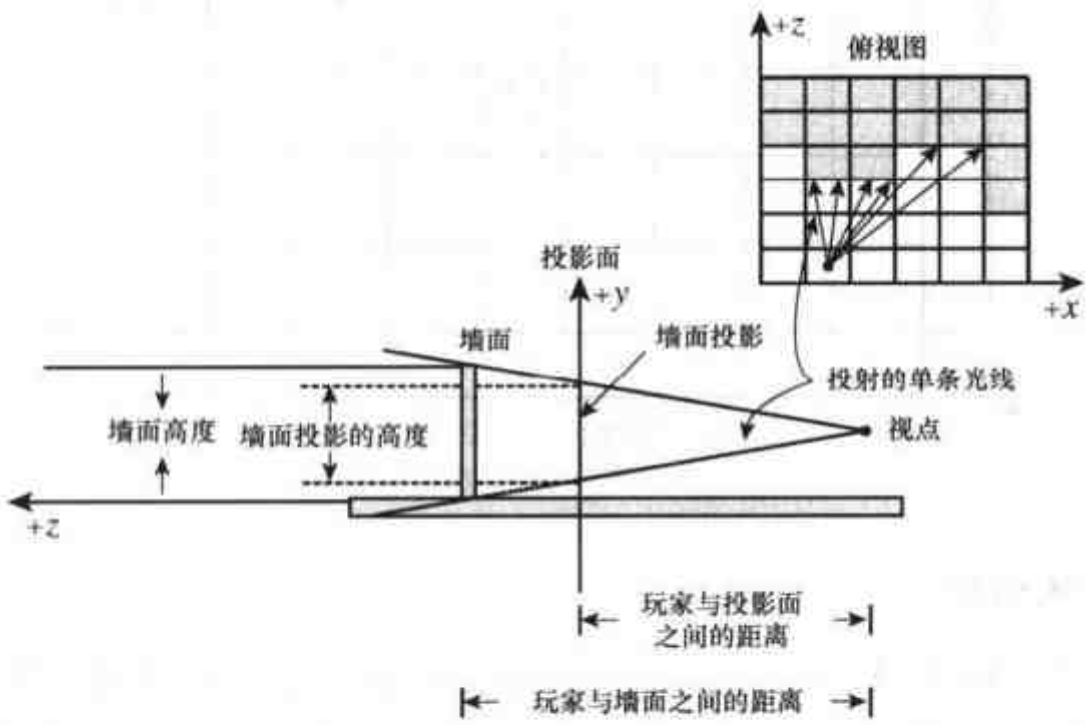


图 6.58　3D 第一人称射击游戏 Unreal Tournament

最后，FPS 的物理模型和 AI 非常复杂，同样其原因在于这种游戏的近距离性质。玩家对周围的一些都看得很清楚，能够预期到将发生的情况。在太空射击游戏中，太空飞船爆炸模拟起来不太难，但当玩家被脉冲能量击中而跌入水中时，玩家希望看到水波，箱子在水中上下漂浮。

6.9.4　光线投射和体素引擎

前面讨论了基于多边形的引擎，但还有其他类型的 3D 引擎，它们是基于光线投射（ray casting）和体素（voxel）的。光线投射是一种被用于很多一流 3D FPS 游戏（如 Wolffenstein 3D）中的技术，这些游戏是基于前向光线跟踪（forward ray tracing）的，即从玩家的视点投射一条光线，穿过视平面，直到遇到物体，如图 6.59 所示。根据游戏世界的几何形状和有关物体的线索，这种技术可以非常快地生成 3D 场景。



$$\frac{\text{墙面投影的高度}}{\text{玩家到投影面的距离}} = \frac{\text{墙面高度}}{\text{玩家到墙面的距离}}$$

根据三角形相似可知：

图 6.59　光线投射机制

注意：实际上，第一个 3D 光线投射程序是由一位名为 Paul Edelstein 的程序员创建的，作者在 80 年代末期曾经在一款 Atari 800 游戏中见到过；但人们很久以后才意识到这种技术的强大威力。

光线投影的问题在于其功能有限。然而，作者相信，在 5 年之内，我们将再次看到高级的光线投射程

序，它们以某种方式混合使用光线投射和光线跟踪，以真实感比多边形引擎更强的方式进行渲染。另一种光线投影技术叫体素图形学（voxel graphics）或体渲染（volumetric rendering）。Novalogic 的 Comandne 是使用体素图形学的最著名的游戏之一。从技术上说，真正的体素图形学并非基于光线投射，而是一种名为行进立方体算法（marching cubes algorithm）的技术，这种算法被用来渲染体数据。每个 3D 图像都是使用一组 3D 数据来渲染的，因此数据和要渲染的图像之间存在一一对应的关系。医学显像技术使用体素系统。然而，我们发现，可以根据体素数据，使用基于光线投射的其他技巧非常快地渲染图像。如果读者运行附带光盘中的体素演示程序，便可知道其速度有多快。

光线投射引擎和体素引擎的问题在于，要表示任何包含细节的东西，都需要大量的数据。在当今的游戏中，这些引擎的应用并不广泛，因为带纹理的多边形的效率更高，视觉效果更好。但正如作者说过的，不久的将来它们将重现江湖，因为归根结底，所有的渲染技术都将是基于光源和光线跟踪的，而光线投射和体素引擎在这些方面很擅长。

6.9.5　混合引擎

这里要介绍的最后一种引擎是混合引擎，这意味着引擎被设计成能够同时模拟太空、陆地和 FPS。这也意味着单个引擎能够在相同的环境中使用不同的技术。例如，Wolfenstein 使用光线投射来渲染游戏世界，用子画面来表示物体；Doom 使用多边形来表示游戏世界，使用子画面来表示物体；Quake 使用多边形来表示游戏世界和有些物体，使用子画面来表示其他物体，在 Quake III 中，一切都是基于多边形的。

关键之处在于，在很多时候，您可能想创建一个有多种环境的游戏，这要求您必须使用多个不同类型的引擎，以便根据要完成的任务使用合适的引擎，而不是使用一个引擎来完成所有的任务。例如，当今的很多 3D 游戏都有粒子系统。很多时候，粒子系统是基于被称为广告牌的多边形的，即这种多边形总是面向观察者——就像是 3D 子画面。

总之，本书将始终使用基于多边形的 3D 引擎。如果读者对其他技术感兴趣——可能是想使游戏中的某些物体更容易实现等，市面上有很多有关子画面引擎、光线投射和其他技术的图书。

6.10　将各种功能集成到引擎中

本章介绍了很多主题，编写了大量的伪代码，列举了很多范例。现在，可以开始实际编写这样的工作代码：让我们能够创建粗糙的 3D 世界，将物体放入其中，移动物体，通过相机观察整个世界等。这些工作都将在下一章完成；随后的每一章都将重点介绍某些特定的主题，并进一步完善我们的引擎。本章编写了一些与物体和数据结构相关的代码，这些代码以及下一章的全部代码都可以在 T3DLIB5.CPP|H 中找到。

本章没有演示程序，只介绍了一些高度抽象的概念，列举了一些范例。从现在开始，作者将假设读者掌握了这些内容，并加快步伐；如果读者没有充分理解它们，请复习本章。

6.11　总　　结

本章无疑介绍了大量的基础知识，读者现在的问题可能比阅读本章前更多。随着读者往下阅读，这些问题将有望得到解答。本章的目标是让读者详细了解坐标系统——因为下一章将直接建立在本章的基础之上，并对 3D 图形学的方方面面有大致的了解——给读者提出一些在详细学习每个主题时需要思考的问题。正如作者说过的，3D 图形学非常复杂，需要以循序渐进的方式进行学习。

第 7 章　渲染 3D 线框世界

现在，可以编写 3D 应用程序了。读者具备了基本的数学知识，并对 3D 引擎的各个组成部分有了总体的认识。本章编写一个简单的 3D 线框引擎，接下来的章节将其升级为恒定着色处理引擎，加入光照、纹理映射，最终得到一个类似于 Quake 的软件引擎。不同于前一章，本章将编写大量的代码，同时对于每项工作，都将介绍两种不同的实现方法。另外，本章还将讨论除欧拉模型外的另一种相机模型——UVN 系统。本章包含以下主题：

- 线框引擎的总体体系结构；
- 编写一个简单的 3D 文件加载器；
- 建立 3D 流水线；
- 局部坐标到世界坐标变换；
- 相机模型；
- 世界坐标到相机坐标变换；
- 相机坐标到透视坐标变换；
- 透视坐标到屏幕视口坐标变换；
- 物体消除；
- 背面消除；
- 简单的 2D/3D 裁剪；
- 渲染 3D 图像。

7.1　线框引擎的总体体系结构

编写第一个 3D 引擎之前，首先简要地介绍一下最终的引擎是什么样的。对作者而言，仅仅提供一个类似于 Quake 的引擎没有任何意义，本书旨在逐步完善引擎，让读者了解从空白开始构建 3D 引擎的每个细节。这包括为构建 3D 引擎所做的错误假设和简化。

提示：说到犯错或过度简化，最典型的例子是原子模型。在高中乃至大学低年级的课程中，使用了这样的原子模型：原子由原子核和环绕圆形或椭圆形轨道运行的电子组成，其中前者由中子和质子组成。当然，情况并非如此。实际上，电子并非绕轨道运行，而是电子云，我们无法同时确定电子的位置和动量。但对于很多简单实验而言，轨道模型是可行的。同样，我们将学习的很多 3D 技术也有改进的空间，如果读者只知道这样做比那样做更好，而不知其所以然，注定将重蹈历史的覆辙，犯类似的错误。

前一章介绍了大量有关 3D 图形学的数学理论。读者已具备构建很不错的引擎所需的全部 3D 图形学知识（一些光栅化技术除外），也许读者还未意识到这一点。现在我们要做的是，使用前面介绍的有关坐标系、投影和数据结构的数学知识和算法，来构建一个简单的 3D 线框引擎。它让您能够以独立的方式或通过加载磁盘文件中的物体来定义多边形，进而渲染它们。另外，还将编写各种变换函数，以便能够创建各种 3D

流水线。例如，可以加载一个物体，并在流水线中使用硬编码对其进行变换，然后将物体作为一组多边形进行渲染。另一方面，也可以使用矩阵来执行各种流水线变换，将每个物体分解成多边形，然后渲染这些多边形，而不关心这些多边形表示的物体。

关键之处在于，将介绍多种实现同一个目标的方式，如果读者是 X Window System 程序员，将对此将有似曾相识之感。读者将发现，3D 公理（如矩阵）虽然很不错，但并不适合用于完成诸如局部坐标到世界坐标变换等任务，至少在软件引擎中是这样的。最后，将探讨两种不同的相机模型。第一种是欧拉模型，它指定了相机的位置和一组旋转角度；第二种是 UVN 系统（只描述，而没有实现），它类似于电影导演的思维方式，指定方向和焦点以及相对于该方向的朝向。

随便说一句，如果读者没有高级 T1（或 Casio）制图计算器，应购买一个，它们将在数学方面给您提供非常大的帮助。另外，还可以在这种计算器中编写游戏，因为它们装备了 32 位的 68000 处理器。

7.1.1　数据结构和 3D 流水线

我们的第一个 3D 引擎涉及的数据结构并不多。构建 3D 引擎之前，先简要地介绍一些数据源和 3D 流水线。图 7.1 说明了该引擎基于的 3 种数据结构：

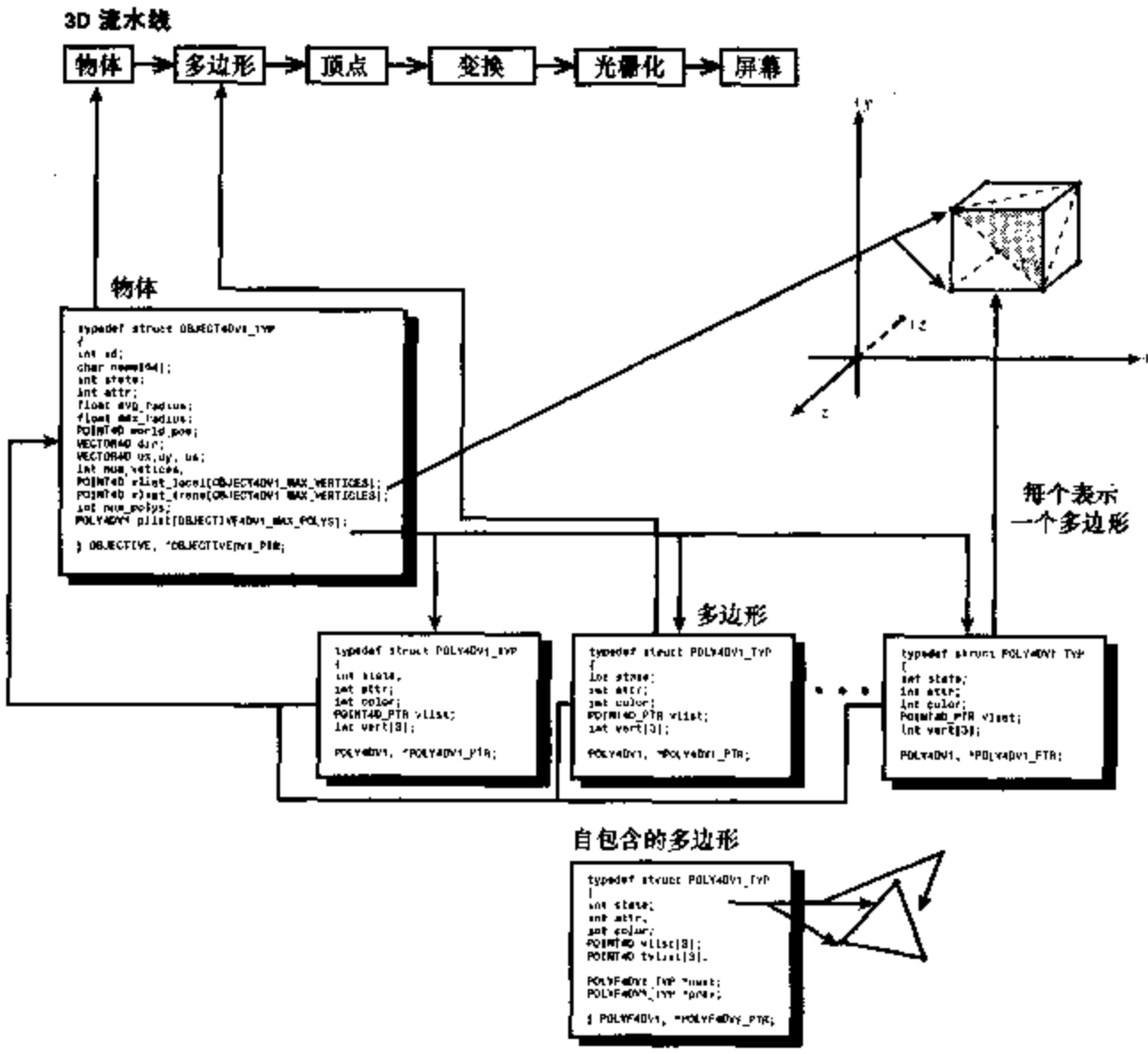


图 7.1　3D 引擎使用的数据结构

- 基于顶点列表的多边形数据结构；
- 自包含的多边形数据结构；
- 表示由多个多边形组成的物体的数据结构。

它们是使用前一章介绍的数据结构实现的。出于温习的考虑，这里再次列出它们：

```
// 基于顶点列表的多边形
typedef struct POLY4DV1_TYP
{
int state;      //   状态信息
int attr;       //   多边形的物理属性
int color;      //   多边形的颜色
```

```
POINT4D_PTR vlist; //顶点列表
int vert[3]; // 顶点列表中元素的索引

} POLY4DV1, *POLY4DV1_PTR;
// 自包含的多边形数据结构，供渲染列表使用
typedef struct POLYF4DV1_TYP
{
int state;  // 状态信息
int attr;   // 多边形的物理属性
int color; // 多边形的颜色

POINT4D vlist[3]; // 该三角形的顶点
POINT4D tvlist[3]; // 变换后的顶点

POLYF4DV1_TYP *next; // 指向渲染列表中下一个多边形的指针
POLYF4DV1_TYP *prev; // 指向渲染列表中前一个多边形的指针

} POLYF4DV1, *POLYF4DV1_PTR;
```

下面是一些为简化工作而使用的#define 语句：

```
// 多边形和多边形面的属性
#define POLY4DV1_ATTR_2SIDED      0x0001
#define POLY4DV1_ATTR_TRANSPARENT    0x0002
#define POLY4DV1_ATTR_8BITCOLOR     0x0004
#define POLY4DV1_ATTR_RGB16       0x0008
#define POLY4DV1_ATTR_RGB24       0x0010

#define POLY4DV1_ATTR_SHADE_MODE_PURE   0x0020
#define POLY4DV1_ATTR_SHADE_MODE_FLAT   0x0040
#define POLY4DV1_ATTR_SHADE_MODE_GOURAUD 0x0080
#define POLY4DV1_ATTR_SHADE_MODE_PHONG  0x0100

// 多边形和面的状态值
#define POLY4DV1_STATE_ACTIVE     0x0001
#define POLY4DV1_STATE_CLIPPED    0x0002
#define POLY4DV1_STATE_BACKFACE    0x0004
```

有些符号看似名不符实，但随着构建引擎的工作逐渐进行下去，读者将明白其含义。下面是表示物体的数据结构。

```
// 基于顶点列表和多边形列表的物体
typedef struct OBJECT4DV1_TYP
{
int id;       // 物体的数字 ID
char name[64];    // 物体的字符名称
int state;   // 物体的状态
int attr;    // 物体的属性
float avg_radius; // 物体的平均半径，用于碰撞检测
float max_radius; // 物体的最大半径

POINT4D world_pos; // 物体在世界坐标系中的位置

VECTOR4D dir;    // 物体在局部坐标系中的旋转角度
          // 用户定义的坐标或单位方向向量

VECTOR4D ux,uy,uz; // 局部坐标轴，用于存储物体的朝向
          // 旋转期间将被自动更新

int num_vertices;  // 物体的顶点数
POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES]; // 用于存储顶点局部坐标的数组
POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES]; // 存储变换后的顶点坐标的数组

int num_polys;    // 物体网格的多边形数
POLY4DV1 plist[OBJECT4DV1_MAX_POLYS]; // 多边形数组

} OBJECT4DV1, *OBJECT4DV1_PTR;
```

注意：如果使用指向多边形和顶点的指针数组，效率将更高，但现在不考虑内存的问题。使用动态数组会使算法更复杂。

图 7.1 是一条简化的 3D 流水线，它接受物体或多边形，将其局部坐标进行变换，最后得到屏幕坐标。下面讨论要使用 3D 引擎来处理什么。

在大多数情况下，3D 引擎接受大量的物体或表示几何体的多边形。在某个地方，所有物体都被转换为多边形，并被加入到渲染（光栅化）列表中。然而，在比较简单的 3D 引擎中，很可能将所有物体原封不动地作为独立的实体进行处理和渲染。在大多数情况下，您不希望在渲染阶段还涉及高级层次型物体。大多数 3D 光栅化器都处理多边形而不是物体，但也可以编写处理物体的光栅化器，只要它能够正确地处理物体。事实上，读者可能认为，在光栅化阶段，可能无法处理物体，因为没有单个多边形列表，其中包含构成场景中各个几何体的所有多边形。那么，如果对多边形进行排序，以便按正确的顺序渲染它们呢？

从理论上说，只要对物体进行排序，然后对构成物体的多边形进行排序，便可以正确地渲染每个物体，因为每个物体都定义了一个凸空间（为进行排序，可将物体视为一个包围它的球体或长方体）。当然，如果有 z 缓存（将在后面讨论），则使用多边形列表还是物体列表无关紧要，因为在这种情况下，光栅化所有物体时，都将对每个像素进行深度测试。

上一段旨在重申我们完成每项工作时都将采用多种方式，以便读者能够明白不同技术的优缺点。因此，我们将编写各种代码：接受单个多边形并对其进行光栅化；接受一组物体，并在光栅化之前一直将它们作为物体；将物体分解为多边形，并将其加入到主多边形列表中。接下来讨论主多边形列表。

7.1.2　主多边形列表

在大多数 3D 引擎（无论是基于软件还是硬件的）中，在某个时候将把游戏中的所有物体从高级表示转换为多边形，并将这些多边形插入到主多边形列表或流（stream）中，然后将主多边形列表传递给 3D 流水线进行处理。高级表示可能是 3D 网格、参数化描述或算法动态生成的。主多边形列表可能是在世界坐标系下生成的，也可能是在流水线的下游生成的，例如，进行相机变换后将所有物体转换为多边形。但无论如何，将把单个多边形列表传递给光栅化器。当然，在我们的 3D 引擎中，也有一个这样的列表，如图 7.2 所示。如果愿意，可以使用它来存储多边形。用于表示第一个多边形列表版本的数据结构是一个指针数组，其中的指针指向静态数组，如图 7.2 所示；其定义如下：

```
typedef struct RENDERLIST4DV1_TYP
{
int state;  // 渲染列表的状态
int attr;   //  渲染列表的属性

// 渲染列表是一个指针数组
// 其中每个指针指向一个自包含的、可渲染的多边形面 (POLYF4DV1)
POLYF4DV1_PTR poly_ptrs [RENDERLIST4DV1_MAX_POLYS];

// 实际多边形数据
POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];

int num_polys;  // 渲染列表中包含的多边形数目

} RENDERLIST4DV1,  *RENDERLIST4DV1_PTR;
```

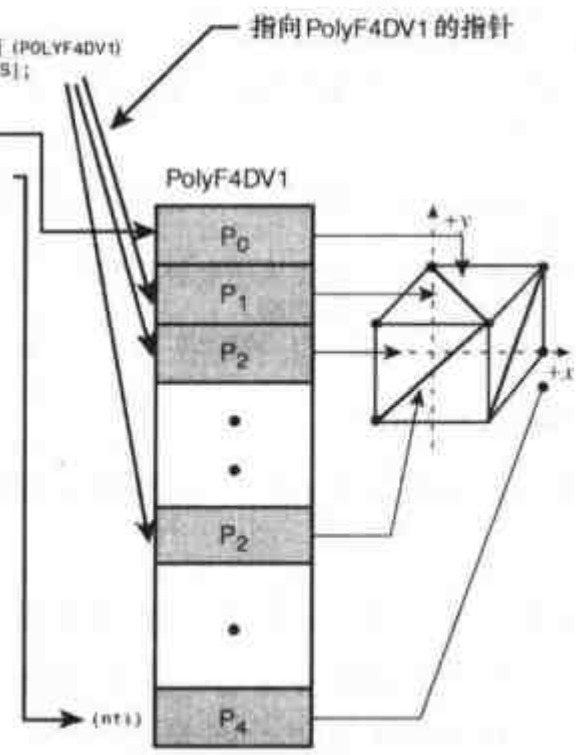


图 7.2　主多边形列表

```
// 存储渲染列表的对象，这样可以同时有多个渲染列表
typedef struct RENDERLIST4DV1_TYP
```

```
{
int state; // 渲染列表的状态
int attr; // 渲染列表的属性

// 渲染列表是一个指针数组
// 其中每个指针指向一个自包含的、可渲染的多边形面（POLYF4DV1）
POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];

// 为避免每帧都为多边形分配和释放存储空间
// 多边形存储在下述数组中
POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];

int num_polys; // 渲染列表中包含的多边形数目

} RENDERLIST4DV1, *RENDERLIST4DV1_PTR;
```

数据结构 RENDERLIST4DV1 包含字段 state 和 attr，它们用于跟踪列表的状态和属性；但最重要的是数组 poly_ptrs[ ]和 poly_data[ ]。这些数组实现了间接列表（索引列表），即数据存储在数组 poly_data[ ]中，但我们不直接遍历数组 poly_data[ ]，而是遍历数组 poly_ptrs[ ]，该数组中包含指向 poly_data[ ]元素的指针或索引。

注意：在 Direct3D 中，这种技术被称为索引顶点缓存（indexed vertex buffer）。

读者可能觉得这种技术有些怪异，但基于以下原因需要使用它。首先，需要根据某种键值（如 z 值，用于排列渲染顺序）对多边形进行排序；为此，可以对数组 poly_ptrs[ ]而不是 poly_data[ ]进行处理。其次，要删除某个多边形时，如果没有数组 poly_ptrs[ ]，必须从数组 poly_data[ ]中将其删除，并移动其他所有的多边形。然而，通过在数组 poly_ptrs[ ]中存储间接指针，只需在该数组中删除相应的指针即可。最后，读者可能会说，为何玩这么多花样，使用一个链表不就行了嘛！确实如此，但就现在而言，使用静态数组更容易；仅当多边形数目在 2 到 200 万之间变化时，才应使用链表。当前，每帧需要处理的多边形最多几百个，因此采用静态分配存储空间的方式，而不使用链表。

最后，在有多部相机时，可能有多个渲染列表，因此作者决定不将渲染列表声明为全局变量。要定义渲染列表，可以这样做：

```
RENDERLIST4DV1 rend_list;
```

然后，调用重置函数，以重置渲染列表，供动画的下一帧使用：

```
void Reset_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list)
{
// 这个函数初始化和重置传递进来的渲染列表
// 为将多边形插入到其中做好准备
// 这个版本的渲染列表由一个 FACE4DV1 指针数组组成
// 每一帧都需要调用这个函数

// 这里使用 num_polys 来跟踪渲染列表中包含的多边形数目
// 因此将其设置为 0
// 如果需要使渲染列表更通用，需要采用更健壮的方案
// 并将其与多边形指针列表的关联切断
rend_list->num_polys = 0; // 硬编码

} // Reset_RENDERLIST4DV1
```

在后面需要创建动态地渲染列表时，该函数将更复杂，因为它可能需要释放内存，并清除前一帧的多边形数据。

注意：在 3D 图形学中，数据结构的重要性怎么强调都不过分，读者必须熟悉链表、双链表、树、图等数据结构以及查找、插入和删除等算法。有关这些数据结构和算法的详细讨论，请参阅 Sedgewick 编写的 *Algorithms in C++*或 Leendert Ammeraal 编写的 *Programs and Data Structure*。

熟悉所需的数据结构后，便可以开始编写 3D 线框引擎了。有趣的是，这很简单，确实如此，只不过是一些细节和关联，就像填写纳税申报单一样简单。需要做的只是创建一个多边形或加载一个物体（需要

一个完成这项任务的函数），然后执行下述步骤：

1. 执行局部坐标到世界坐标变换；
2. 物体消除（可选）；
3. 执行背面消除（可选）；
4. 执行世界坐标到相机坐标变换；
5. 执行 3D 裁剪（可选）；
6. 执行相机坐标到透视坐标变换，即投影；
7. 执行透视坐标到视口（屏幕）坐标变换；
8. 光栅化多边形。

注意：有时候第 6 步和第 7 步被合而为一。

当然，有些步骤的顺序可能与上面列出的不同，另外，要处理物体还是将其转换为多边形（将在流水线的某个地方执行这种转换）也有关系。最后，可以选择采用矩阵、手工方法或混合方法来执行变换。例如，使用矩阵乘法来执行局部坐标到世界坐标变换无疑是浪费处理器周期，因为如果这样做，对于每个点需要执行 16 次乘法运算和 12 次加法运算，而这本来只需执行 3 次加法运算便可完成。另一方面，相机变换几乎总是使用矩阵来完成的，因为相机变换矩阵不是稀疏矩阵。这里要说的是，没有放之四海而皆准的解决之道，速度快且管用的方法就是正确的方法。然而，本章将在完成每项工作时都采用两种不同的方式，让读者对可在流水线中使用的各种方法有深入了解，同时知道有些方法是及其低效的——尤其是使用齐次坐标时。使用齐次坐标有利也有弊。

7.1.3　新的软件模块

本章新增的库模块如下，它们包含了 3D 线框引擎的所有功能代码：

- **T3DLIB5.H**——头文件；
- **T3DLIB5.CPP**——C/C++源代码。

当然，要编译本章中的程序，需要包含之前的所有 T3DLIB*.CPP|H 文件以及所有 DirectX 库，还需要要求编译器生成 Win32 .EXE，读者应该知道如何完成这些任务。另外，在每个演示程序的开头，作者使用注释指出了为编译程序所需的文件；如果读者不知道需要哪些库文件，可查看注释。

警告：今天，作者收到 13 封指出编译程序时出现链接错误 error main() no found 的邮件。除非特别指明是 DOS 控制台应用程序，否则所有 Windows Win32/DirectX 应用程序都是 Win32 .EXE，入口为 WinMain()。因此，出现上述编译错误的惟一原因是，读者将 Windows 应用程序作为 DOS CONSOLE 应用程序进行编译，并相应地设置编译器。作者打算建立一个 CAGA（美国编译器滥用小组）的讨论小组，专门针对那些编译器产品滥用者；如果读者对此感兴趣，可给我发邮件，地址为 compilerabuser@xgames3d.com <GRIN>。

7.2　编写 3D 文件加载器

前一章介绍了很多 3D 文件格式，读者可能安装了附带光盘中的或自己购买的 3D 建模程序，并将诸如立方体等简单物体导出，在 ASCII 编辑器中查看其数据。如果读者选修过有关编译器和分析器设计方面的课程，编写 3D 文件阅读器将是小菜一碟；如果不是这样，编写语言分析器将是一件苦差事，因为这种任务确实不那么好对付。必须对自己要做的事情有深入认识。如果要完成大量的 3D 工作，可能需要有一套 3D 文件导入/导出/转换函数，这样就不用自己去编写它们。尽管如此，作者喜欢自力更生，将自己编写一个本书要使用的简单文件阅读器，而不使用第三方的产品。作者将尽可能使之简短、清晰、切中要害。

为此，本书将首先使用文件格式.PLG 和.NFF，因为它们易于理解，作者将在今晚 Sci-Fi 频道播出 *first wave* 之前编写好这两种文件的分析程序，这是一个重要的考虑因素。因为，这两种文件格式都是文本文件，

读者可手工编写它们，本章将这样做。为何这样做呢？旨在让读者对如何处理 3D 坐标有所认识，然后再将这项工作交给 3D 建模程序去处理。

首先要编写的是.PLG 分析程序，但这里不详细介绍.PLG 文件格式，因为前一章已经介绍过；而只是概要地描述这种文件是什么样的。这种文件格式是基于顶点列表的，它包含一个多边形列表，多边形是使用顶点索引号来定义的。.PGL 文件类似于这样：

```
# 这是注释
# 首行为物体名称、顶点数和多边形数
object_name num_vertices num_polygons

# 顶点列表
# 每个顶点的格式都为 x y z
x0 y0 z0
x1 y1 z1
x2 y2 z2
x3 y3 z3
.
.
xn yn zn

# 多边形列表
# 每个多边形的格式如下
surface_descriptor num_vertices v0 v1 . vn
surface_descriptor num_vertices v0 v1 . vn
surface_descriptor num_vertices v0 v1 . vn
surface_descriptor num_vertices v0 v1 . vn
.
.
surface_descriptor num_vertices v0 v1 . vn
# 其中 v0 v1 . vn 是多边形的顶点的编号
```

下面是一个描述立方体的.PLG 文件：

```
# plg/plx 文件的开头

# 简单立方体，8 个顶点，12 个多边形
tri 8 12

# 顶点列表，顶点坐标按 x、y、z 顺序排列
 5 5 5
-5 5 5
-5 5 -5
 5 5 -5
 5 -5 5
-5 -5 5
-5 -5 -5
 5 -5 -5

# 多边形列表
# 每个多边形有 3 个顶点
0xd0f0 3 0 1 2
0xd0f0 3 0 2 3
0xd0f0 3 0 7 4
0xd0f0 3 0 3 7
0xd0f0 3 4 7 6
0xd0f0 3 4 6 5
0xd0f0 3 1 6 2
0xd0f0 3 1 5 6
0xd0f0 3 3 6 7
0xd0f0 3 3 2 6
0xd0f0 3 0 4 5
0xd0f0 3 0 5 1
```

PLG 文件存在的惟一问题是，它是 20 世纪 80 年代的产品，因此对其进行了修改，添加一些功能，并将修改后的格式称为.PLX 文件。关键之处在于，如果读者找到了能够导出.PLG 文件的 3D 建模程序，我们

将编写的阅读器仍能够读取这种文件，但 16 位的面描述符将是无效的。在修改后的.PLG 文件（作者有时将其称为.PLX 文件）中，面描述符的定义如下：

```
CSSD | RRRR | GGGG | BBBB
```

其中：

- C 为 RGB/索引颜色标记；
- SS 描述了着色模式；
- D 为单面标记；
- RRRR、GGGG 和 BBBB 为 RGB 模式下红色、绿色和蓝色分量；
- GGGGBBBB 为 8 位颜色索引模式下的颜色索引。

下面的#define 用于识别并抽取这些位字段：

```
// 用于简化测试的位掩码
#define PLX_RGB_MASK     0x8000// 抽取 RGB/索引颜色模式的掩码
#define PLX_SHADE_MODE_MASK 0x6000// 抽取着色模式的掩码
#define PLX_2SIDED_MASK    0x1000   // 抽取双面状态设置的掩码
#define PLX_COLOR_MASK    0x0fff  // xxxxrrrrggggbbbb，每个 RGB 分量 4 位
                        // xxxxxxxxiiiiiiii，颜色索引模式下的 8 位索引

// 用于判断颜色模式的标记
#define PLX_COLOR_MODE_RGB_FLAG   0x8000     // 多边形使用 RGB 颜色
#define PLX_COLOR_MODE_INDEXED_FLAG 0x0000 // 多边形使用 8 位颜色索引

// 双面标记
#define PLX_2SIDED_FLAG       0x1000 // 多边形是双面的
#define PLX_1SIDED_FLAG       0x0000 // 多边形是单面的

// 着色模式标记
#define PLX_SHADE_MODE_PURE_FLAG  0x0000  // 多边形使用固定颜色
#define PLX_SHADE_MODE_FLAT_FLAG  0x2000  // 恒定着色
#define PLX_SHADE_MODE_GOURAUD_FLAG 0x4000 // gouraud 着色
#define PLX_SHADE_MODE_PHONG_FLAG  0x6000  // phong 着色
```

例如，要定义一个使用 RGB 颜色模式（红、绿、蓝颜色分量分别是 0x3、0x8 和 0xF）和 Gouraud 着色的双面多边形，可以按如下方式来计算面描述符。

鉴于 16 位的面描述符格式如下：

```
CSSD | RRRR | GGGG | BBBB
```

且：

C=1 表示 RGB 颜色模式；

SS=10 表示 Gouraud 着色；

D=1 表示双面多边形；

因此半字节 CCSD 为 1101；

半字节 RRRR、GGGG 和 BBBB 分别为 0x3、0x8 和 0xF，因此面描述符为：

```
1101 0011 1000 1111 = 0xD38F
```

面描述符的前 4 位定义了多边形的属性和颜色模式，余下的 12 位指出 RGB 模式下的颜色或颜色索引模式下的索引。文件阅读器需要做的只是将这些位的值提取出来，并在存储物体的数据结构中设置相应的值。下面介绍加载.PLG 文件的函数。

.PLG/X 文件加载器

.PLG 文件的优点之一是，其格式是固定的，也就是说顶点在前，多边形在后，而不能相反；而.DXF 文件的格式要灵活些。.PLG 文件的每行都包含一个数字或多个用空格分隔的数字。因此，可以编写一个简单的分析程序和一个粗糙的有限状态机，前者读取文件中的每一行，而后者知道.PLG 文件分 3 部分：

- 文件头信息；
- 顶点数据；
- 多边形数据。

每部分的格式都是固定的，且数据是基于行的（line-base data），这意味着每行数据都不会延续到下一行。因此，可以这样编写一个每次读取一行数据的分析程序，然后根据分析程序的状态（当前位于哪部分）分析和提取数据元素（它们由空格隔开），仅此而已。当然，分析程序分析数据的同时，将结果插入到某种数据结构（这里为 OBJECT4DV1）中。

编写该分析程序时，可以使用一个函数，但这样做代码将非常难看，需要在该函数中判断各种条件，并读取数据行。作者决定将其分成两个函数。

提示：确定分析方法正确后，将首先编写一组分析函数，然后使用它们来编写阅读器。当然，也可以使用商用工具，如 YACC 和 LEX，它们都是基于 UNIX 的工具（也有 PC 版本），可用于编写编译器和解释器。

第一个支持函数 Get_Line_PLG() 从.PLG 文件中读取一行数据（使用了函数 fgets()），如下所示：

```
char *Get_Line_PLG(char *buffer, int maxlength, FILE *fp)
{
// 这个辅助函数跳过 PLG 文件中的注释和空行
//返回一整行数据
//如果文件为空，则返回 NULL

int index = 0; // 索引
int length = 0; // 长度

// 进入分析循环
while(1)
   {
   // 读取下一行
   if (!fgets(buffer, maxlength, fp))
   return(NULL);

   // 计算空格数
   for (length = strlen(buffer), index = 0;
      isspace(buffer[index]); index++);

   // 检查是否是空行或注释
   if (index >= length || buffer[index] == '#')
     continue;
   // 此时得到了一个数据行
   return(&buffer[index]);
   } // end while

} // end Get_Line_PLG
```

这个函数很简单。它识别空格以及用#标记的注释，并返回文件中的下一行，如果文件为空，则返回 NULL。后一种情况仅当出现错误时才会发生，因为.PLG 文件中包含头信息行指定的数据量。实际读取.PLG 文件的函数名为 Load_OBJECT4DV1_PLG()（请注意这里的命名规则），如下所示：

```
int Load_OBJECT4DV1_PLG(OBJECT4DV1_PTR obj, // 指向物体的指针
          char *filename,       // plg 文件的名称
          VECTOR4D_PTR scale,   // 缩放因子的初始值
          VECTOR4D_PTR pos,     // 初始位置
          VECTOR4D_PTR rot)     // 初始旋转角度
{
// 这个函数从磁盘中加载 plg 物体
// 并允许调用程序对物体进行缩放和旋转以及设置其位置
// 以免再调用其他函数来设置非动态物体的这些参数

FILE *fp;      // 文件指针
char buffer[256]; // 缓冲区

char *token_string; // 指向要分析的物体数据文本的指针
```

```
// 复习 PLG 文件的格式
// # 这是注释

// # 物体描述符
// object_name_string num_verts_int num_polys_int

// #顶点列表
// x0_float y0_float z0_float
// x1_float y1_float z1_float
// x2_float y2_float z2_float
// .
// .
// xn_float yn_float zn_float
//
// # 多边形列表
// surface_description_ushort num_verts_int v0_index_int v1_index_int .
// vn_index_int
// .
// .
// surface_description_ushort num_verts_int v0_index_int v1_index_int .
// vn_index_int

// 假设每个元素占一行
// 因此必须找到物体描述符，读取它
// 然后找到并读取顶点列表，最后找到并读取多边形列表

//第 1 步：清空和初始化 obj
memset(obj, 0, sizeof(OBJECT4DV1));

//将物体状态设置为可见和活动的
obj->state = OBJECT4DV1_STATE_ACTIVE | OBJECT4DV1_STATE_VISIBLE;

// 设置物体的位置
obj->world_pos.x = pos->x;
obj->world_pos.y = pos->y;
obj->world_pos.z = pos->z;
obj->world_pos.w = pos->w;

//第 2 步：为读取而打开文件
if (!(fp = fopen(filename, "r")))
   {
   printf("Couldn't open PLG file %s. ", filename);
   return(0);
   } // end if

// 第 3 步:读取物体描述符
if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
   {
   Write_Error("PLG file error with file %s (object descriptor invalid). ",filename);
   return(0);
   } // end if

Write_Error("Object Descriptor: %s", token_string);

// 对物体描述符进行分析
sscanf(token_string, "%s %d %d", obj->name,
                &obj->num_vertices,
                &obj->num_polys);
// 第 4 步：加载顶点列表
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
   {
   // 读取下一个顶点
   if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
   {
    Write_Error("PLG file error with file %s (vertex list invalid). ",
          filename);
```

```
    return(0);
    } // end if

// 分析顶点
sscanf(token_string, "%f %f %f", &obj->vlist_local[vertex].x,
                &obj->vlist_local[vertex].y,
                &obj->vlist_local[vertex].z);
obj->vlist_local[vertex].w = 1;

// 缩放顶点坐标
obj->vlist_local[vertex].x*=scale->x;
obj->vlist_local[vertex].y*=scale->y;
obj->vlist_local[vertex].z*=scale->z;

Write_Error("\nVertex %d = %f, %f, %f, %f", vertex,
                    obj->vlist_local[vertex].x,
                    obj->vlist_local[vertex].y,
                    obj->vlist_local[vertex].z,
                    obj->vlist_local[vertex].w);

} // end for vertex

// 计算平均半径和最大半径
Compute_OBJECT4DV1_Radius(obj);

Write_Error("\nObject average radius = %f, max radius = %f",
      obj->avg_radius, obj->max_radius);

int poly_surface_desc = 0; // PLG/PLX 多边形描述符
int poly_num_verts  = 0; // 当前多边形的顶点数（总是 3）
char tmp_string[8];    // 存储多边形描述符的字符串

// 第 5 步：加载多边形列表
for (int poly=0; poly < obj->num_polys; poly++)
  {
// 读取多边形描述符
if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
   {
   Write_Error("PLG file error with file %s (polygon descriptor invalid).",
         filename);

   return(0);
   } // end if

Write_Error("\nPolygon %d:", poly);

// 每个多边形都有 3 个顶点
// 因为假定所有模型都是由三角形组成的
// 将面描述符、顶点数和顶点列表存储到变量中
sscanf(token_string, "%s %d %d %d %d", tmp_string,
                &poly_num_verts, // 总是 3
                &obj->plist[poly].vert[0],
                &obj->plist[poly].vert[1],
                &obj->plist[poly].vert[2]);

// 面描述符可以是十六进制值（以 0x 打头）
// 因此需要对这种情况进行检测
if (tmp_string[0] == '0' && toupper(tem_string[1]) == 'X')
  sscanf(tmp_string, "%x", &poly_surface_desc);
else
  poly_surface_desc = atoi(tmp_string);

// 让多边形顶点列表指向物体的顶点列表
// 这是多余的，因为多边形列表包含在物体中
// 建立多边形几何体时，由用户决定使用局部顶点列表还是变换后的顶点列表
// 如果所有多边形都属于同一个物体，将该指针设置为 NULL 将更合适
obj->plist[poly].vlist = obj->vlist_local;
```

```
Write_Error("\nSurface Desc = 0x%.4x, num_verts = %d,
        vert_indices [%d, %d, %d]",
                        poly_surface_desc,
                        poly_num_verts,
                        obj->plist[poly].vert[0],
                        obj->plist[poly].vert[1],
                        obj->plist[poly].vert[2]);
// 存储顶点列表和多边形顶点索引值后
// 分析多边形描述符，并据此相应地设置多边形

// 提取多边形描述符中的每个位字段
// 从单面/双面位开始
if ((poly_surface_desc & PLX_2SIDED_FLAG))
   {
   SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_2SIDED);
   Write_Error("\n2 sided. ");
   } // end if
else
   {
   // 单面
   Write_Error("\n1 sided.");
   } // end else

// 设置颜色模式
if ((poly_surface_desc & PLX_COLOR_MODE_RGB_FLAG))
   {
   // 为 RGB 颜色模式
   SET_BIT(obj->plist[poly].attr,POLY4DV1_ATTR_RGB16);

   // 提取 RGB 颜色值
   int red  = ((poly_surface_desc & 0x0f00) >> 8);
   int green = ((poly_surface_desc & 0x00f0) >> 4);
   int blue = (poly_surface_desc & 0x000f);
   // 虽然在文件中，RGB 颜色总是为 4.4.4 格式，而图形卡
   // 的 RGB 颜色格式为 5.5.5 或 5.6.5，但虚拟颜色系统将把
   //8.8.8 颜色格式转换为 5.5.5 或 5.6.5
   //因此需要将 4.4.4 的 RGB 颜色值转换为 8.8.8 的
   obj->plist[poly].color = RGB16Bit(red*16, green*16, blue*16);
   Write_Error("\nRGB color = [%d, %d, %d]", red, green, blue);
   } // end if
else
  {
   // 使用的是 8 位颜色索引
   SET_BIT(obj->plist[poly].attr,POLY4DV1_ATTR_8BITCOLOR);
   // 提取最后 8 位即可得到颜色索引
   obj->plist[poly].color = (poly_surface_desc & 0x00ff);

   Write_Error("\n8-bit color index = %d", obj->plist[poly].color);

   } // end else

// 处理着色模式
int shade_mode = (poly_surface_desc & PLX_SHADE_MODE_MASK);

// 设置多边形的着色模式
switch(shade_mode)
    {
    case PLX_SHADE_MODE_PURE_FLAG: {
    SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_PURE);
    Write_Error("\nShade mode = pure");
    } break;

    case PLX_SHADE_MODE_FLAT_FLAG: {
    SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_FLAT);
    Write_Error("\nShade mode = flat");
```

```
        } break;

        case PLX_SHADE_MODE_GOURAUD_FLAG: {
        SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_GOURAUD);
        Write_Error("\nShade mode = gouraud");
        } break;

        case PLX_SHADE_MODE_PHONG_FLAG: {
        SET_BIT(obj->plist[poly].attr, POLY4DV1_ATTR_SHADE_MODE_PHONG);
        Write_Error("\nShade mode = phong");
        } break;

        default: break;
        } // end switch

    // 最后将多边形设置为活动状态
    obj->plist[poly].state = POLY4DV1_STATE_ACTIVE;

    } // end for poly

    // 关闭文件
    fclose(fp);
    // 指出操作成功
    return(1);

    } // end Load_OBJECT4DV1_PLG
```

该函数中包含大量的错误检查代码，这可能有些过度，但旨在让读者养成在加载文件的函数中使用错误处理代码的习惯，因为这种函数很容易出错，且将数据写入内存（如果仅仅是初始化变量，则无需这样做）。因此，在游戏代码中，作者从来不使用错误处理代码，因为如果代码不完善，我可以改进。但在文件加载函数中，指望外部文件是正确的，但情况可能并非如此。因此，当加载函数加载数据时，如果发生错误，应将错误消息发送给错误文件（即标准输出设备——译者注）。

接下来介绍这个函数的功能。它打开字符串 ***filename*** 指定的.PLG 文件，然后读取文件中的第一行，这一行指出了文件中包含多少个顶点和多边形的数据。然后，该函数读取顶点列表，将其中的顶点插入到 OBJECT4DV1 结构 obj 中；再读取并插入多边形数据，同时将面描述符中的标记和颜色模式转换为 POLY_ATTR_*标记，并使用 OR 运算将它们组合起来。另外，还可以给这个函数传递一些参数，以便在加载物体的同时对其进行变换；这些参数是 scale、pos 和 rot，它们都是向量，分别指定了物体的初始位置、缩放因子和旋转角度。

另外，这个函数还计算物体的平均半径和最大半径，供 3D 流水线的后面进行碰撞检测和物体消除时使用。要从磁盘中加载物体数据，可以这样做：

```
OBJECT4DV1 obj; // object storage

VECTOR4D scale = {1,1,1,1},   // 不缩放
     pos   = {0,0,0,1},      // 放置到世界坐标系原点处
     rot   = {0,0,0,1};      // 不旋转

// 加载物体
Load_OBJECT4DV1_PLG(&obj, "cube.plg", &scale, &pos, &rot);
```

如果使用调试器遍历数据结构 obj，读者将发现物体数据已被存储到其中且是正确的（但愿如此）。上述将外部文件加载到数据结构 OBJECT4DV1 中的操作初始化物体；设置字段 state 和 attr；将定义物体初始朝向的 ux、uy 和 uz 分别设置为<1，0，0，1>、<0，1，0，1>和<0，0，1，1>，即分别与世界坐标系的各个轴平行。

至此，有 3 个可以使用的数据结构：

- POLY4DV1——表示一个多边形，但需要引用外部顶点列表；
- POLYF4DV1——表示一个多边形，内部存储了顶点数据，是主多边形列表的主要组成部分；
- OBJECT4DV1——表示一个由众多多边形组成的物体，内部存储了顶点和多边形，即是自包含的。

可以编写引擎，在流水线中处理这些数据结构，但作者选择只对 POLYF4DV1 和 OBJECT4DV1 进行处理，因为 POLY4DV1 是没有顶点列表的 POLYF4DV1。下面开始编写引擎，在流水线中对多边形和像素进行处理。

7.3　构建 3D 流水线

现在可以开始实现 3D 流水线了。数据结构定义好了，从.PLG 文件中加载物体的函数也有了，只需要编写一些辅助（darn）函数即可。

警告：惟一遗憾的地方是，编写在屏幕上绘制图像的演示程序之前，需要编写大量的函数，因此这些函数之间是彼此相关的。但编写好这些函数后，作者将通过大量的演示程序来演示流水线中渲染阶段前的各种技术。

7.3.1　通用变换函数

在 3D 中的任何变换都可以通过 4×4 矩阵变换以及 4D 齐次坐标到 3D 坐标转换（将各个分量除以 w）来完成。然而，这并不意味着这种方法的速度是最快的。一般而言，对于由旋转、平移和投影组合而成的变换，使用一个 4×4 矩阵来完成更合适，因为这个矩阵涵盖了多种变换，通常不是稀疏矩阵。然而，如果仅仅是平移一个点，使用矩阵乘法就有宰牛刀杀鸡之嫌，因为将矩阵和点相乘时，需要执行 16（4*4）次乘法运算和 12（4*3）次加法运算；而使用手工来执行平移时，只需 3 次加法运算：

```
x=x+xt;
y=y+yt;
z=z+zt;
w=w;
```

在编写 3D 引擎时，读者可能发现，手工方法又快又简单。然而，虽然如此，但代码不那么清晰，也容易移植到基于硬件的引擎中，在基于硬件的引擎中，几乎所有操作都是使用 4×4 矩阵来完成的。因此，在我们的引擎中，只要可能总是允许使用这两种方法。

为使用 4×4 矩阵来执行多边形或物体变换，需要编写一个这样的函数，它将多边形（POLYF4DV1）或物体（OBJECT4DV1）作为参数，并使用一个变换矩阵对几何体的每个顶点执行矩阵乘法。另外，还需要考虑到这两个数据结构中都包含用于存储原始顶点坐标和变换后顶点坐标的数组。

本书前面介绍过，一旦对物体的顶点数据进行变换，原来的数据就将丢失。因此，必须确信确实要这样做，在很多情况下确实如此；但在有些情况下，变换是流水线的一部分，因此需要保留原始数据不动，将变换后的数据存储到另一个数组中。数据结构 POLYF4DV1 和 OBJCET4DV1 中都有存储这种数据的数组。

在数据结构 POLYF4DV1 中，顶点数据存储在下列两个数组中：

```
POINT4D vlist[3]; // 三角形的顶点

POINT4D tvlist[3];   // 变换后的三角形顶点
```

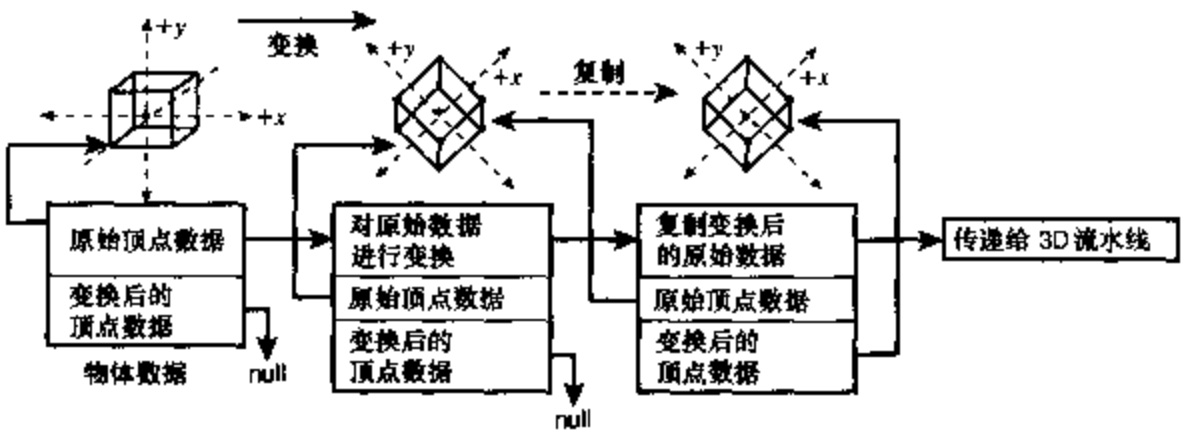
其中 tvlist[ ]用于存储变换后的数据。同样，在数据结构 OBJECT4DV1 中，顶点数据存储在下述数组中：

```
POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES]; // 存储局部顶点坐标的数组

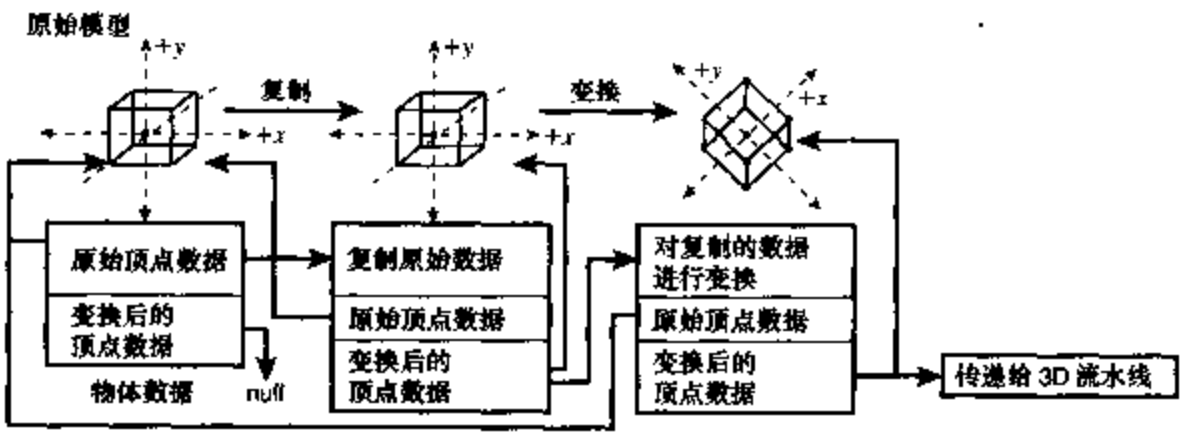
POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES]; //存储变换后顶点坐标的数组
```

因此，如果愿意可以删除第二个数组中的数据，因为第一个数组中存储了原始数据。然而，有时候需要对原始数据进行变换（可能为实现动画而进行缩放或旋转），并存储变换后的数据。在这种情况下，可以对存储原始数据的数组元素进行修改。图 7.3 是对顶点数据执行的两种操作的流程图。

需要编写一个使用变换矩阵对 POLYF4DV1 进行变换的函数。但大多数情况下，不会对单个多边形进行变换，而是等到有了包含所有多边形的列表后，再进行变换。因此我们从变换多边形列表的角度来考虑问题，如主多边形列表，这是用结构 RENDERLIST4DV1 表示的，这种结构是一个 POLYF4DV1 数组。下面是一个使用矩阵 mt 对渲染列表进行变换的函数：



A. 破坏性变换



B. 非破坏性变换

图 7.3　对顶点数据的破坏性变换和非破坏性变换流程图

```
void Transform_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
        MATRIX4X4_PTR mt,  // 变换矩阵
        int coord_select)  // 指定要变换的坐标
{
// 这个函数使用传递进来的矩阵
// 对渲染列表中局部顶点数组或变换后的顶点数组中
// 所有的多边形顶点进行变换

// 应对哪个数组中的坐标进行变换?
switch(coord_select)
{
case TRANSFORM_LOCAL_ONLY:
{
for (int poly = 0; poly < rend_list->num_polys; poly++)
   {
  // 获得当前多边形
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // 这个多边形有效吗?
  // 当且仅当多边形处于活动状态并可见时才对其进行变换
  // 在线框引擎中，"背面"的概念无关紧要
  if ((curr_poly==NULL) ||
     !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
     (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
     (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形

  // 满足条件，进行变换
  for (int vertex = 0; vertex < 3; vertex++)
    {
    // 使用矩阵 mt 对顶点进行变换
    POINT4D presult; // 用于暂时存储变换结果

    // 对点进行变换
    Mat_Mul_VECTOR4D_4X4(&curr_poly->vlist[vertex], mt, &presult);

    // 将结构存回去
    VECTOR4D_COPY(&curr_poly->vlist[vertex], &presult);
```

```
    } // end for vertex
  } // end for poly

} break;

case TRANSFORM_TRANS_ONLY:
{
// 对渲染列表中每个变换后的顶点进行变换
// 数组 tvlist[]用于存储累积变换结果
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // 获得当前多边形
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];
  // 该多边形有效吗?
  // 当且仅当多边形处于活动状态并可见时才对其进行变换
  // 在线框引擎中，"背面"的概念无关紧要
  if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
     (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
     (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
     continue; // 进入下一个多边形

  // 满足条件，开始变换
  for (int vertex = 0; vertex < 3; vertex++)
    {
    // 使用矩阵 mt 对顶点进行变换
    POINT4D presult; // 用于存储变换结果

    // 变换
    Mat_Mul_VECTOR4D_4X4(&curr_poly->tvlist[vertex], mt, &presult);
    // 将结果存回去
    VECTOR4D_COPY(&curr_poly->tvlist[vertex], &presult);
    } // end for vertex
  } // end for poly

} break;

case TRANSFORM_LOCAL_TO_TRANS:
{
// 对渲染列表中的局部/模型顶点列表进行变换
// 并将结果存储到变换后的顶点列表中
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // 获得下一个多边形
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // 该多边形有效吗?
  // 当且仅当多边形处于活动状态并可见时才对其进行变换
  // 在线框引擎中，"背面"的概念无关紧要
  if ((curr_poly==NULL) ||
       !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
     (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
     (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形
    // 满足条件，开始变换
   for (int vertex = 0; vertex < 3; vertex++)
     {
     // 使用矩阵 mt 对顶点进行变换
     Mat_Mul_VECTOR4D_4X4(&curr_poly->vlist[vertex], mt,
                          &curr_poly->tvlist[vertex]);
   } // end for vertex

  } // end for poly

} break;

default: break;
```

```
} // end switch

} // end Transform_RENDERLIST4DV1
```

这个函数非常简单。它根据坐标变换选择标记 coord_select，将 4×4 矩阵 mt 应用于渲染列表：通过循环和调用矩阵乘法函数对 coord_select 指定的坐标进行变换。参数 coord_select 的可能取值如下：

```
// 变换控制标记
#define TRANSFORM_LOCAL_ONLY   0 // 对局部/模型顶点列表进行变换
#define TRANSFORM_TRANS_ONLY   1 // 对变换后的顶点列表进行变换
#define TRANSFORM_LOCAL_TO_TRANS 2 // 对局部顶点列表进行变换
                    // 并将结果存储在变换后的顶点列表中
```

因此，1 表示对局部坐标进行变换，2 表示对变换后的坐标进行变换，而 3 表示将局部坐标进行变换，并将结果存储为变换后的坐标。

这个函数的用法很简单：如果有一个名为 rend_list 的渲染列表，其中包含所有的多边形，要使用变换矩阵 m_trans 对其中每个多边形的局部坐标进行变换，并将结果作为变换后的坐标，可以这样调用该函数：

```
Transform_RENDERLIST4DV1(&rend_list, m_trans, TRANSFORM_LOCAL_TO_TRANS );
```

这样，将使用上述 4×4 矩阵对整个渲染列表进行变换。

接下来编写的是对物体（OBJECT4DV1）进行变换的函数，它与前一个函数基本相同，但更简单些，因为它只变换一个顶点列表，而不是一组各自都有一个顶点列表的多边形。当然，在这个函数中，仍然需要使用参数来指定要对哪种坐标进行变换。该函数的代码如下：

```
void Transform_OBJECT4DV1(OBJECT4DV1_PTR obj, // 要变换的物体
         MATRIX4X4_PTR mt,  // 变换矩阵
         int coord_select,  // 指定对哪种坐标进行变换
         int transform_basis) // 指定是否要对朝向向量进行变换
{
// 这个函数使用传递进来的矩阵
// 对局部数组或变换后的数组中的所有顶点进行变换

// 对哪种坐标进行变换?
switch(coord_select)
    {
    case TRANSFORM_LOCAL_ONLY:
    {
    // 对物体的每个局部/模型顶点坐标进行变换
    for (int vertex=0; vertex < obj->num_vertices; vertex++)
      {
      POINT4D presult; // 用于暂时存储变换结果

      // 对顶点进行变换
      Mat_Mul_VECTOR4D_4X4(&obj->vlist_local[vertex], mt, &presult);

      // 将结果存回去
      VECTOR4D_COPY(&obj->vlist_local[vertex], &presult);
      } // end for index
    } break;

    case TRANSFORM_TRANS_ONLY:
    {
    // 对物体的每个变换后的顶点进行变换
    // 数组 vlist_trans[]用于存储累积变换结果
    for (int vertex=0; vertex < obj->num_vertices; vertex++)
      {
      POINT4D presult; // hold result of each transformation

      // 对顶点进行变换
      Mat_Mul_VECTOR4D_4X4(&obj->vlist_trans[vertex], mt, &presult);

      // 将结果存回去
      VECTOR4D_COPY(&obj->vlist_trans[vertex], &presult);
      } // end for index
```

```
    } break;

    case TRANSFORM_LOCAL_TO_TRANS:
    {
    // 将物体的每个局部/模型顶点坐标进行变换
    // 并将结果存储到变换后的顶点列表中
    for (int vertex=0; vertex < obj->num_vertices; vertex++)
      {
      POINT4D presult; // hold result of each transformation

      // 对顶点进行变换
      Mat_Mul_VECTOR4D_4X4(&obj->vlist_local[vertex], mt,
      &obj->vlist_trans[vertex]);

      } // end for index
    } break;

    default: break;

} // end switch

// 最后检查是否要对朝向向量进行变换
// 如果不进行变换，朝向向量将不再有效
if (transform_basis)
   {
   // 旋转物体的朝向向量
   VECTOR4D vresult; // 用于存储旋转结果

   // 旋转 ux
   Mat_Mul_VECTOR4D_4X4(&obj->ux, mt, &vresult);
   VECTOR4D_COPY(&obj->ux, &vresult);
   // 旋转 uy
   Mat_Mul_VECTOR4D_4X4(&obj->uy, mt, &vresult);
   VECTOR4D_COPY(&obj->uy, &vresult);
   // 旋转 uz
   Mat_Mul_VECTOR4D_4X4(&obj->uz, mt, &vresult);
   VECTOR4D_COPY(&obj->uz, &vresult);
   } // end if

} // end Transform_OBJECT4DV1
```

同样，在这个函数中，变换方式也有三种，但还有一个 transform_basis 标记。使用这个标记的原因如下：如图 7.4 所示，对 3D 物体进行变换时，将丢失其加载时的朝向。也就是说，如果您知道物体被加载时，上方为+Y 轴方向，前方为+Z 轴方向，如何记录物体被变换后的朝向呢？

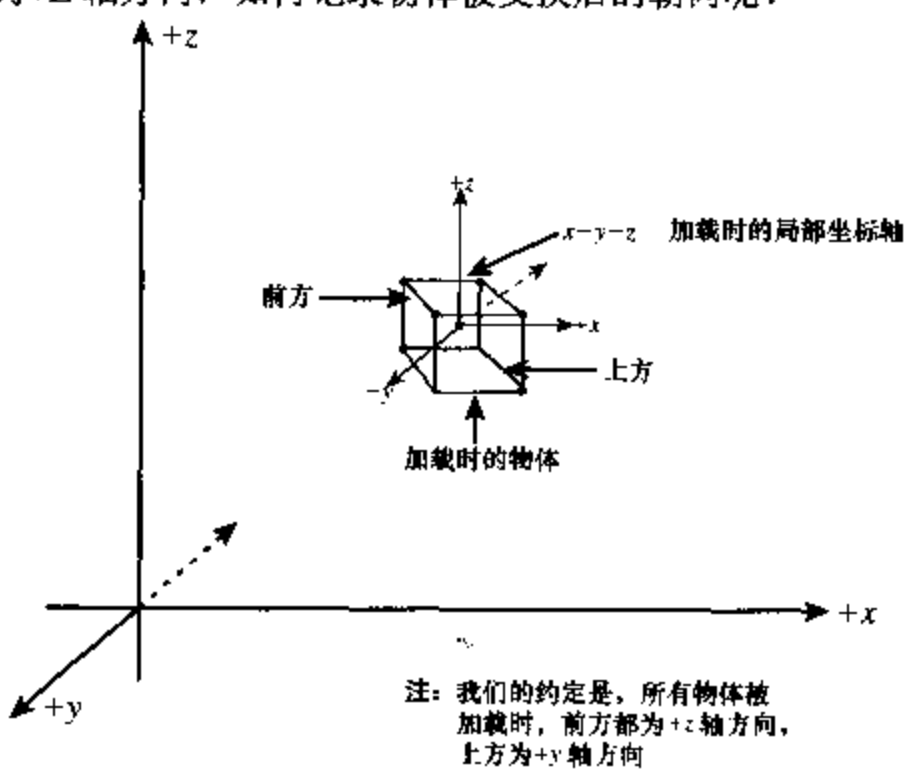


图 7.4　物体被加载时的朝向

关键之处在于，也需要对物体的局部坐标轴进行变换。这样，总是可以通过将物体的局部坐标轴同世界坐标轴进行比较，来确定物体的朝向；这是一个很常见的问题，作者收到了很多有关这一问题的邮件。图 7.5 说明了同时对物体的顶点数据和局部坐标轴进行变换的情况。标记 transform_basis 指出了是否要对物体的局部坐标轴也进行变换。物体的局部坐标轴可以为任何初始值，但通常 ux = <1，0，0，0>，uy = <0，1，0，0>，uz = <0，0，1，0>，即与实际坐标轴平行。

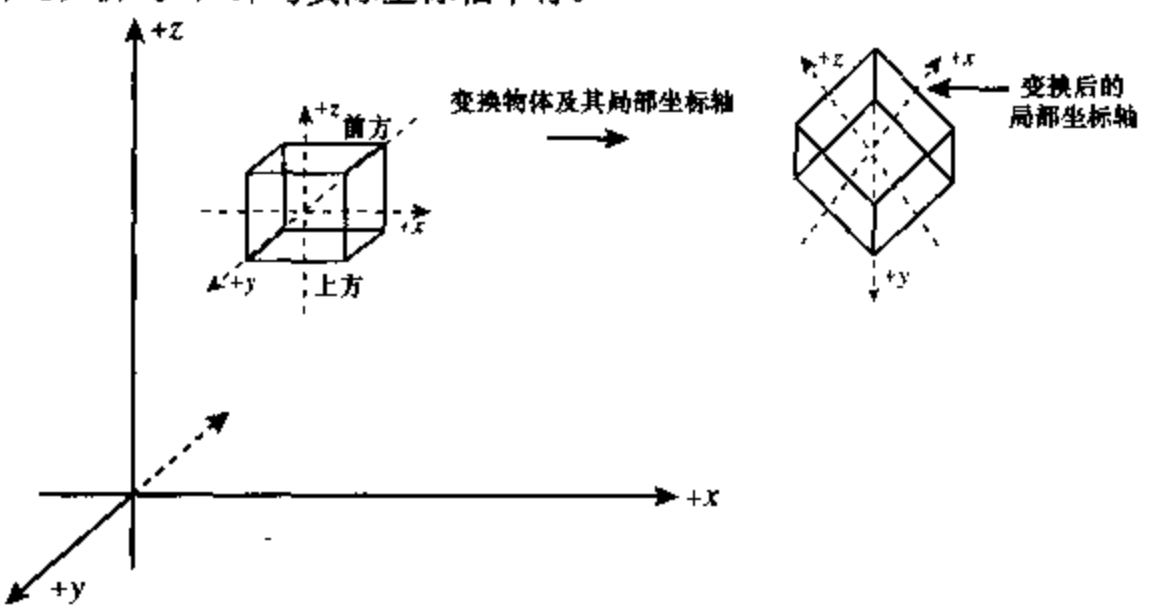


图 7.5　对局部坐标轴进行变换以跟踪物体的朝向

例如，假设有一个已被加载到变量 obj 中的物体，要使用矩阵 m_trans 对其局部坐标和局部坐标轴进行变换，可以这样调用上述函数：

```
Transform_OBJECTDV1(&obj, m_trans, TRANSFORM_LOCAL_TO_TRANS, 1 );
```

和偷税一样简单！当然，作者不会这样干。

有了通用的变换函数后，便可以将一个 4×4 矩阵作为参数来调用该函数，以实现任何变换。然而，别忘了，对于很多变换而言，手工计算的速度更快。接下来的几节将介绍如何执行流水线中的各个步骤：使用手工方法和矩阵方法对 OBJECT4DV1 和 RENDERLIST4DV1 进行变换。

7.3.2　局部坐标到世界坐标变换

局部坐标到世界坐标变换将物体（多边形）的局部/模型坐标变换为世界坐标（整个游戏空间是使用世界坐标定义的），如图 7.6 所示。给定点/顶点 **p** = <*x*，*y*，*z*，1>和平移因子 **dt** = <*xt*，*yt*，*zt*，1>，局部坐标到世界坐标变换很简单：

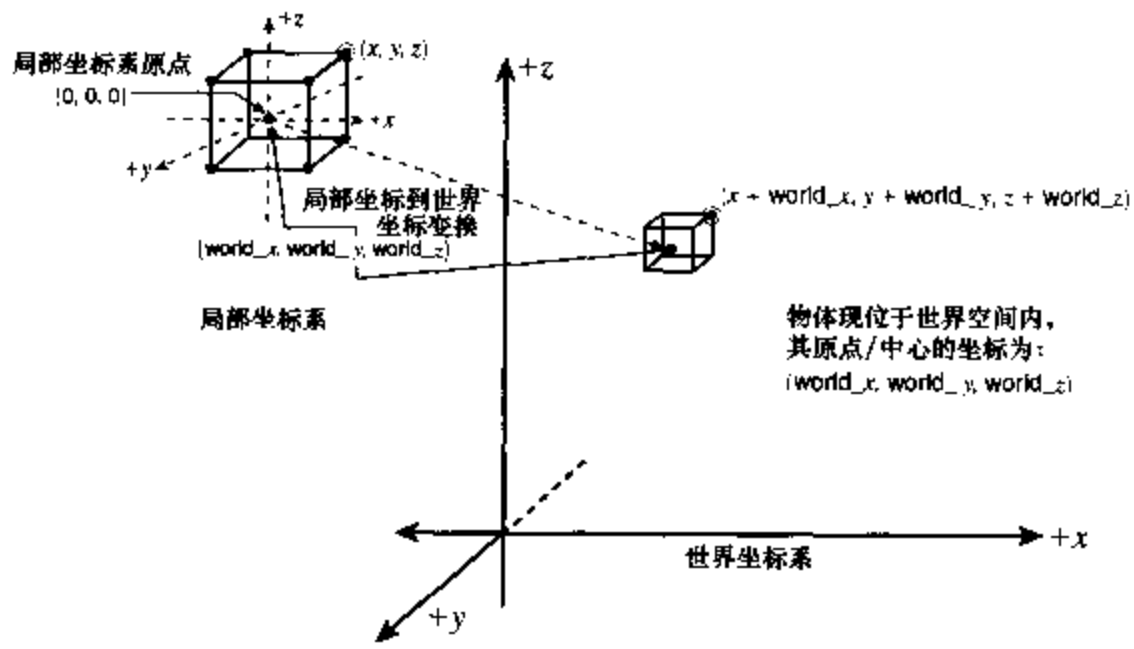


图 7.6　局部坐标到世界坐标变换

$$\mathbf{p}' = \mathbf{p} + \mathbf{dt}$$

其中分量 w 保持不变，仍为 1。

1. 物体的局部坐标到世界坐标变换

对 OBJECT4DV1 的执行局部坐标到世界坐标变换的函数根据 world_pos 的值，将存储在 local_vlist[ ] 中的顶点进行平移，并将结果存储在 trans_vlist[ ]中。下面是这个函数的手工计算版本：

```
void Model_To_World_OBJECT4DV1(OBJECT4DV1_PTR obj,
         int coord_select= TRANSFORM_LOCAL_TO_TRANS)
{
// 这里没有使用矩阵
// 这个函数将传递进来的物体的局部/模型坐标变换为世界坐标
// 结果被存储在物体的变换后的顶点列表（数组 vlist_trans）中

// 遍历顶点列表，根据 world_pos 对所有坐标进行平移
// 以便将坐标变换为世界坐标
// 并将结果存储在数组 vlist_trans[ ]中

if (coord_select == TRANSFORM_LOCAL_TO_TRANS)
{
for (int vertex=0; vertex < obj->num_vertices; vertex++)
  {
  // 对顶点进行平移
  VECTOR4D_Add(&obj->vlist_local[vertex], &obj->world_pos,
         &obj->vlist_trans[vertex]);
  } // end for vertex
} // end if local
else // TRANSFORM_TRANS_ONLY
{
for (int vertex=0; vertex < obj->num_vertices; vertex++)
  {
  // 平移顶点
  VECTOR4D_Add(&obj->vlist_trans[vertex], &obj->world_pos,
         &obj->vlist_trans[vertex]);
  } // end for vertex
} // end else trans

} // end Model_To_World_OBJECT4DV1
```

假设已经将一个单位立方体加载到一个名为 cube 的 OBJECT4DV1 结构中，并将其 world_pos 设置为（100，200，400），要对其进行变换，可以这样调用上述函数：

```
Model_To_World_OBJECT4DV1(&cube);
```

该函数有一个用于指定要对哪种坐标进行变换的参数，这在需要对变换后的坐标再进行其他变换时很有用，为此，可以使用参数 coord_select 进行控制。

如果愿意，也可以使用矩阵来执行这种变换，但这样做将浪费 CPU 周期。然而，我们必须介绍这种方法，因为硬件喜欢使用矩阵。建立变换矩阵的函数为 void Build_Model_To_World_MATRIX4X4 ()，如下所示：

```
void Build_Model_To_World_MATRIX4X4(VECTOR4D_PTR vpos, MATRIX4X4_PTR m)
{
// 这个函数创建局部坐标到世界坐标变换矩阵
// 即根据 vpos 指定的值平移原点

Mat_Init_4X4(m, 1,    0,    0,    0,
        0,    1,    0,    0,
        0,    0,    1,    0,
        vpos->x, vpos->y, vpos->z, 1 );

} // end Build_Model_To_World_MATRIX4X4
```

该函数将变换后的位置和用于存储变换矩阵的矩阵作为参数。当然，该函数可用于生成通用平移矩阵，而不仅仅是局部坐标到世界坐标变换矩阵。要使用矩阵来执行局部坐标到世界坐标变换，可以调用上述函数来生成一个矩阵，然后调用函数 Transform_OBJECT4DV1 () 来执行变换，如下所示：

```
MATRIX4X4 mt; // 用于存储创建的矩阵
VECTOR4D pos = {100,200,300, 1};
```

```
Build_Model_To_World_MATRIX4X4(&pos, &mt);

Transform_OBJECT4DV1(&obj, m_trans, TRANSFORM_LOCAL_TO_TRANS,1 );
```

这就是对 OBJECT4DV1 执行局部坐标到世界坐标变换的全部步骤。

注意：这很简单，但从此读者将养成良好的习惯——相信我！

2. 渲染列表的局部坐标到世界坐标变换

大多数情况下，在对物体执行局部坐标到世界坐标变换后，才将物体的多边形插入到渲染列表中。因此，根本不需要对整个渲染列表执行局部坐标到世界坐标变换。但当您不想使用 OBJECT4DV1，而是将多边形直接加载到表示某些大型网格（如地形）的 RENDERLIST4DV1 时，且网格不是用世界坐标表示的时，将需要执行这个步骤。下面是采用手工方法对 RENDERLIST4DV1 执行局部坐标到世界坐标变换的函数：

```
void Model_To_World_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                POINT4D_PTR world_pos, int
                coord_select = TRANSFORM_LOCAL_TO_TRANS)
{
// 这里没有使用矩阵
// 这个函数将传递进来的渲染列表的局部/模型坐标变换为世界坐标
// 结果被存储在渲染列表的变换后的顶点列表（数组 vlist_trans）中

// 遍历顶点列表，根据 world_pos 对所有坐标进行平移
// 以便将坐标变换为世界坐标
// 并将结果存储在数组 vlist_trans[]中

if (coord_select == TRANSFORM_LOCAL_TO_TRANS)
  {
  for (int poly = 0; poly < rend_list->num_polys; poly++)
    {
    // 获得当前多边形
    POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

    // 当且仅当多边形没有被剔除和裁剪掉，同时处于活动状态且可见时
    // 才对其进行变换
    // 但在线框引擎中，多边形是否是背面无关紧要
    if ((curr_poly==NULL) ||
      !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
      (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
      (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
     continue; // 进入下一个多边形

   // 满足条件，对其进行变换
   for (int vertex = 0; vertex < 3; vertex++)
     {
     // 平移顶点
     VECTOR4D_Add(&curr_poly->vlist[vertex], world_pos,
                  &curr_poly->tvlist[vertex]);
     } // end for vertex

   } // end for poly
} // end if local
else // TRANSFORM_TRANS_ONLY
{
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // 获得当前多边形
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // 当且仅当多边形没有被剔除和裁剪掉，同时处于活动状态且可见时
  // 才对其进行变换
  // 但在线框引擎中，多边形是否是背面无关紧要
 if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
    (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
```

```
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
     continue; // 进入下一个多边形

   for (int vertex = 0; vertex < 3; vertex++)
     {
     // 平移顶点
     VECTOR4D_Add(&curr_poly->tvlist[vertex], world_pos,
           &curr_poly->tvlist[vertex]);
     } // end for vertex

   } // end for poly

} // end else

} // end Model_To_World_RENDERLIST4DV1
```

如果有一个名为 rend_list 的渲染列表，要对其执行局部坐标到世界坐标变换，以便将几何体放置到位置 world_pos 处，可以这样做：

```
Model_To_World_RENDERLIST4DV1(&rend_list, &world_pos, TRANSFORM_LOCAL_TO_TRANS);
```

上述函数的效率非常高。要使用矩阵进行上述变换，需要创建一个矩阵，如下所示：

```
MATRIX4X4 mt; // 用于存储创建的矩阵
VECTOR4D pos = {100,200,300, 1};

Build_Model_To_World_MATRIX4X4(&pos, &mt);
```

然后，使用通用函数 RENDERLIST4DV1 对几何体进行变换，如下所示：

```
Transform_RENDERLIST4DV1(&rend_list, &mt, TRANSFORM_LOCAL_TO_TRANS);
```

注意：对每个顶点都需要执行一次矩阵乘法，这样的开销会浪费 CPU 周期。

7.3.3　欧拉相机模型

这并非 3D 流水线中的步骤，但此时是讨论相机模型的最佳时机，因为流水线的后续步骤依赖于相机模型。在我们的第一个系统中，使用的是欧拉相机模型，这种模型是使用位置和旋转角度（欧拉角度）定义的，其中旋转角度决定了相机的朝向，如图 7.7 所示。本章后面还将介绍 UVN 相机模型，它是使用注视目标以及向量 u、v 和 n 定义的。

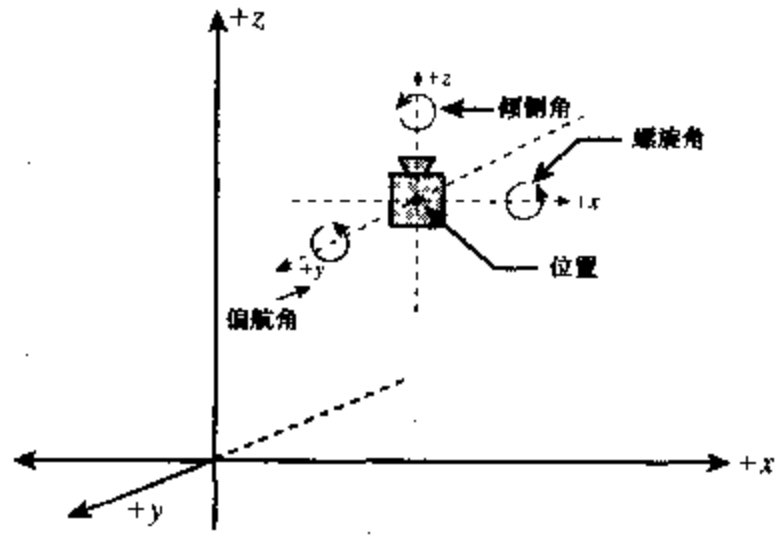


图 7.7　欧拉相机模型

无论使用哪种相机模型，都需要视距、视野、远近裁剪面、一定宽度和高度的视平面（表示要在屏幕上显示的最终图像）以及一定宽度和高度的视口（表示实际的光栅窗口）。这些参数都将在相机模型中发挥作用，它们定义了一个包围相机视野的视景体及其属性。位于视野中的物体是可见的，将对其进行变换。

接下来创建一种用于表示简单相机的结构。本可以不考虑角度、位置、远近裁剪面以及其他几个参数，但为让读者尽早考虑相机需要的所有信息（即使是不使用的信息），这里创建了一个包罗万象的结构。其中有些信息在本章还用不上，尤其是与裁剪相关的信息。相机结构如下：

```
// 第 1 个版本的相机数据结构
typedef struct CAM4DV1_TYP
{
int state;   // 相机状态
int attr;    // 相机属性

POINT4D pos;  // 相机在世界坐标系中的位置

VECTOR4D dir;  // 欧拉角度或 UVN 相机模型的注视方向
```

```
VECTOR4D u;   // UVN 相机模型的朝向向量
VECTOR4D v;
VECTOR4D n;
POINT4D target; // UVN 模型的目标位置


float view_dist_h; // 水平视距和垂直视距
float view_dist_v;

float fov;     // 水平方向和垂直方向的视野

// 3d 裁剪面
// 如果视野不是 90 度，3d 裁剪面方程将为一般性平面方程
float near_clip_z;   // 近裁剪面
float far_clip_z;   // 远裁剪面

PLANE3D rt_clip_plane; // 右裁剪面
PLANE3D lt_clip_plane; // 左裁剪面
PLANE3D tp_clip_plane; // 上裁剪面
PLANE3D bt_clip_plane; // 下裁剪面

float viewplane_width; // 视平面的宽度和高度
float viewplane_height; // 对于归一化投影，为 2×2
            // 否则大小与视口或屏幕窗口相同

// 屏幕和视口是同义词
float viewport_width;   // 屏幕/视口的大小
float viewport_height;
float viewport_center_x; // 视口的中心
float viewport_center_y;

// 宽高比
float aspect_ratio;   // 屏幕的宽高比

// 是否需要下述矩阵取决于变换方法
// 例如，以手工方式进行透视变换、屏幕变换时，不需要这些矩阵
// 然而提供这些矩阵提高了灵活性

MATRIX4X4 mcam; // 用于存储世界坐标到相机坐标变换矩阵
MATRIX4X4 mper; // 用于存储相机坐标到透视坐标变换矩阵
MATRIX4X4 mscr; // 用于存储透视坐标到屏幕坐标变换矩阵

} CAM4DV1, *CAM4DV1_PTR;
```

虽然上述代码中有很多注释，但为避免混淆，下面介绍其中的每个元素：

- state——记录相机的状态，这没有什么可解释的。
- attr——定义相机的属性。当前只有两种属性——CAM_MODEL_ELUER 和 CAM_MODEL_UVN。
- pos——相机在世界坐标系下的位置。
- dir——定义欧拉相机模型的轴线角；使用 UVN 相机模型时，x 分量为球面坐标系的仰角，分量 y 为方向角。
- u、v、n——在 UVN 相机模型（将在本章后面讨论）中，用于记录相机朝向的向量。
- target——UVN 相机模型的注视目标的位置。
- view_list_h、view_dist_v——水平视距和垂直视距，在透视变换中需要使用它们。
- fov——水平视野和垂直视野。
- near_clip_z、far_clip_z——近裁剪面和远裁剪面的 z 坐标值。
- rt_clip_plane、lt_clip_plane、tp_clip_plane、bt_clip_plane——定义左、右、上、下 3D 裁剪面。视野为 90 度时，这些裁剪面从原点出发，与坐标轴呈 45 度角；但在视野为任意值时，这些裁剪面方程不再是 $x=z$、$y=z$ 等，而是广义平面方程。
- viewplane_width、viewplane_height——定义视平面的大小，执行透视投影时，物体将被投影到这

个平面上。对于归一化投影，视平面为 2×2；将透视变换和屏幕变换合而为一时（速度更快），视平面的大小与视口（屏幕窗口）相同。

- viewport_width、viewport_height——光栅面的大小（单位为像素），图像将被渲染到这个面上。
- viewport_center_x、viewport_center_y——视口的中心位置。
- aspect_ratio：viewport_width 和 viewport_height 的比值。
- mcam——用于存储世界坐标到相机坐标变换的矩阵。
- mper——用于存储相机坐标到透视坐标变换的矩阵。
- mscr——用于存储透视坐标到屏幕坐标变换的矩阵。

包含的信息很多，但当前并不需要所有这些信息。创建这种结构旨在存储所有的相机信息，但还有很多派生信息，我们不想定义或计算它们。

接下来需要编写一个接受输入并创建相机的函数。下面是这个函数的原型：

```
void Init_CAM4DV1(CAM4DV1_PTR cam,    // 相机对象
          int cam_attr,      // 相机属性
          POINT4D_PTR cam_pos,  // 相机的初始位置
          VECTOR4D_PTR cam_dir, // 相机的初始角度
          POINT4D_PTR cam_target, // UVN 相机的初始目标位置
          float near_clip_z,   // 近裁剪面和远裁剪面
          float far_clip_z,
          float fov,        // 视野，单位为度
          float viewport_width, // 屏幕/视口的大小
          float viewport_height);
```

可以传递相机的类型、位置、朝向、裁剪面位置、视野、视口的大小，然后该函数将创建一个相机。现在，惟一的问题是，很多参数是相关的，它们会彼此约束。

因此，作者编写该函数时，将确保它创建的视平面的 x 坐标范围为−1 到 1，y 坐标范围为−1/aspect_ratio 到 1/aspect_ratio。必须确定视平面的坐标范围，因为在透视坐标到屏幕坐标变换期间，需要知道如何映射。采用这种约定使得透视坐标到屏幕坐标变换更容易。

在大多数情况下，本书将使用 90 度的视野，这使得视距为 1.0，但作者想在相机设置函数中加入其他功能，让读者能够使用不为 90 度的视野。最后，作者决定在初始化阶段，不应在相机函数中计算相机变换矩阵。相机变换矩阵应在以后通过调用欧拉或 UVN 相机支持（helper）函数来创建。下面介绍 UVN 相机模型。

7.3.4　UVN 相机模型

在很多情况下，欧拉相机模型是可行的，但它有很大的局限性，在指定观察角度时，它并不按您认为的那样工作。UVN 是一种更自然的相机模型，它使用起来容易得多，接下来介绍这种相机模型，并推导相机变换矩阵。图 7.8 是标准的 UVN 系统。

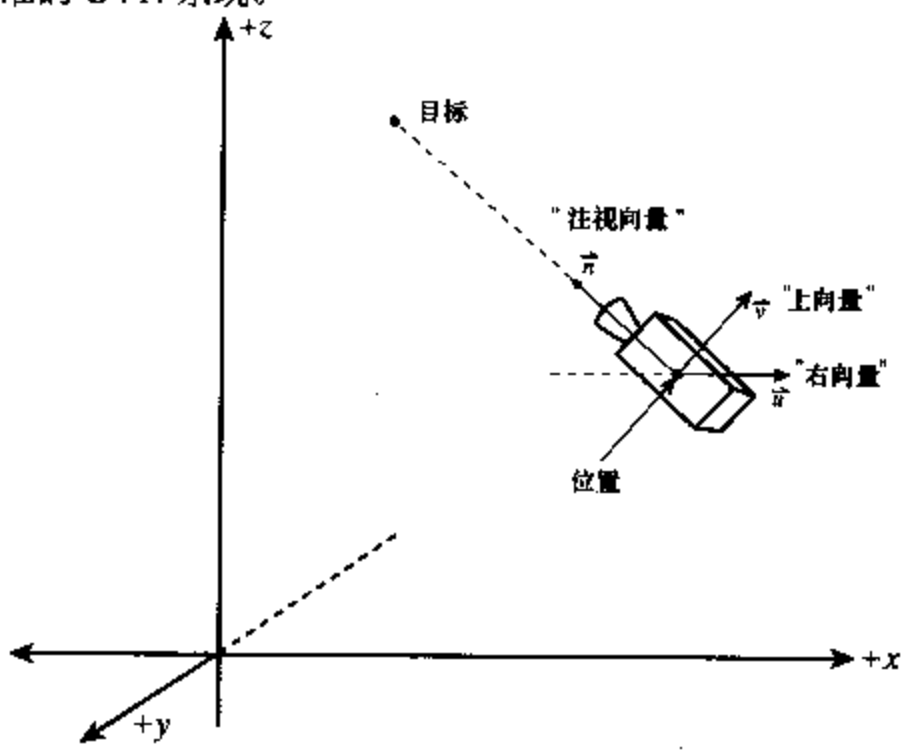


图 7.8　UVN 相机模型

从本质上说，欧拉相机和 UVN 相机之间的惟一差别在于如何定义相机朝向，前者使用角度，而后者使用向量。这些向量定义了相机的朝向（局部坐标轴）：

```
UVN = { X:<ux, uy, uz>, Y:<vx, vy, vz>, Z:<nx, ny, nz>}
```

例如，采用上述表示方法时，标准 XYZ 左手坐标系的坐标轴如下：

```
XYZ = { X:<1, 0, 0>, Y:<0, 1, 0>, Z:<0, 0, 1>}
```

UVN 系统与 XYZ 系统具有相同的性质：所有向量(*U*、*V*、*N*)之间是相互垂直（正交）的，它们的长度为 1 个单位，彼此之间是线性无关的。

另外，UVN 系统还有一个目标（注视点），它指定了相机的观察方向。当然，相机还有空间位置，这被称为观察参考点（view reference point，VRP）。

接下来讨论 UVN 向量及其含义，这里稍微调换了一下顺序。

- *n*——这个向量类似于 UVN 系统的 *Z* 轴，它从相机位置指向目标位置。换句话说，*n* = <目标位置 – VRP>。当然，*n* 必须是单位向量，因此我们将在某个时候对其进行归一化。
- *v*——这个向量被称为上向量，其初始值为 *y* 轴向量<0，1，0>。然而，这个 *v* 值是暂时性的，只用于计算“右”向量 *u*，然后将重新计算 *v*。
- *u*——这个向量被称为右向量，它是 *v* 和 *n* 的叉积——$u = v \times n$，这是临时的 *v* 值发挥作用的地方。计算向量 *u* 后，将使用公式 $v = n \times u$ 重新计算 *v*。

注意：当然，必须在某个时候对所有这些向量进行归一化。

计算出 UVN 系统的 *u*、*v* 和 *n* 向量后，接下来介绍它们的用途。

实际上很简单。UVN 指定了 XYZ 系统的新朝向，所有物体都将被变换到这种空间中，换句话说，它指定了系统的坐标轴。因此，我们需要创建一个执行这种变换的矩阵。也就是说，假设 UVN 相机位于某个位置，并对准某个目标，执行这种变换的矩阵是什么样的呢？首先，暂时不管相机位置，因为它只是一种简单平移而已，复杂的是旋转部分。

1. 计算 UVN 矩阵

要将标准 XYZ 系统中的点 **p**(*x*，*y*，*z*)变换为 UVN 系统中的点，只需分别计算 **p**(*x*，*y*，*z*)与 *u*、*v* 和 *n* 的点积即可。换句话说，需要将 *x* 乘以 *u* 轴和标准 XYZ 坐标系的 *x* 轴之间的共线程度（colinear amount），将 *y* 乘以 *v* 轴和 *y* 轴的共线程度，将 *z* 乘以 *n* 轴和 *z* 轴的共线程度。这样点 **p**(*x*，*y*，*z*)便被变换为 UVN 空间中的点。这个 4×4 矩阵如下：

$$\mathbf{M}_{uvn} = \begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

是不是很简单？这个矩阵的列向量为 *u*、*v* 和 *n*。对于点 **p**(*x*，*y*，*z*，1)，使用上述矩阵对其进行变换时，变换结果 **p**'如下：

$$\mathbf{p}' = \mathbf{p} * \mathbf{M}_{uvn}$$

$$= [x\ y\ z\ 1] * \begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= (x*ux + y*uy + z*uz,\ x*vx + y*vy + z*vz,\ x*nx + y*ny + z*nz,\ 1)$$

表示为点积时，结果如下：

$$\mathbf{p}' = (p.u,\ p.v,\ p.n)$$

问题的最后一部分是相机的位置，这很简单。我们知道，任何相机变换都是平移加旋转，即

$$\mathbf{T}_{cam}^{-1} * \mathbf{R}_{cam}^{-1}$$

其中 $\mathbf{R}_{cam}^{-1}$ 是一种或多种旋转变换的组合。

注意：这里的逆矩阵表明，在原来的推导中，我们获得了放置相机所需的矩阵。要变换到相机空间，需要使用这些矩阵的逆矩阵。旋转的情况与此类似。

前面已经推导出了旋转矩阵 $\mathbf{M}_{uvn}$，还需要推导平移矩阵。假设相机位置为 cam_pos=（cam_x, cam_y, cam_z），则平移矩阵如下：

$$\mathbf{T}_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

因此，UVN 的变换矩阵如下：

$$\mathbf{T}_{uvn} = \mathbf{T}_{cam}^{-1} * \mathbf{M}_{uvn}$$

$$= \begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

$$= \begin{bmatrix} ux & vx & nx & 0 \\ uy & vy & ny & 0 \\ uz & vz & nz & 0 \\ -(cam\_pos \bullet u) & -(cam\_pos \bullet v) & -(cam\_pos \bullet n) & 1 \end{bmatrix}$$

其中最后一行中的各项是平移因子（tx、ty 和 tz），它们分别是相机位置与向量 u、v、n 的点积求负。

至此对 UVN 相机系统的介绍就结束了。我们简要地复习一下，然后介绍如何计算和描述 UVN 系统。

UVN 系统也使用位置来描述相机，但不同的是它指定了一个目标。另外，表示相机时，UVN 系统不使用绕各个轴旋转的角度，而是使用注视向量 n、上向量 v 和右向量 u。这些向量通过下述映射关系同标准 XYZ 轴相关联：

- U -> X
- V -> Y
- N -> Z

计算向量 u、v 和 n 的步骤如下：

第 1 步　n = <目标位置 – 观察参考点>；

第 2 步　假设 v = <0，1，0>；

第 3 步　$u = v \times n$；

第 4 步　$v = n \times u$；

第 5 步　（可选）将向量 u、v 和 n 分别除以其长度，将其归一化（如果之前没有这样做）。

2. UVN 系统的实现

正如读者看到的，UVN 系统很棒，与实际生活中使用相机的方式极其接近：选定相机位置，然后将其对准要拍摄的目标，如图 7.8 所示。存在的惟一问题是，很多时候，并不知道目标的准确坐标，而只知道大概方向。换句话说，在这种情况下，欧拉相机模型更合适，但正如前面指出的，欧拉相机模型并非总能按我们希望的那样做，且存在万向节死锁和其他问题。如果知道注视点，便可以直接计算出向量 u、v 和 n。然而，更自然的基于球面坐标系的相机系统如何呢？在这种系统中，首先需要指定相机的位置，然后指定仰角（elevation）和方位角（heading），如图 7.9 所示。在这种系统中，如何计算 u、v 和 n 呢？这并没有看起来的那么难。请看图 7.9，其中说明了计算步骤。

首先，假设相机位于原点处（被一个单位球面包围），以简化推导过程。显然，要将相机移到任何地方，只需在平移中使用加法运算即可。在图 7.9 中，方位角 θ 是绕+Y 轴旋转的角度，与+Z 轴重叠时为 0 度。仰

角 Φ是方向向量与地平面（x-z 平面）之间的夹角，如图 7.9 所示。现在，在球面坐标系中进行讨论，点的球面坐标为 $\mathbf{p}(\rho, \Phi, \theta)$。然而，在第 4 章的推导中，作者假定为右手坐标系，并假定了一些有关角度的约定；为避免得到与第 4 章完全相同的公式，对假定做细微的修改，如下所示：

$$r = p * \sin \phi$$
$$y = p * \cos \phi$$
$$z = r * \cos \theta$$
$$x = -r \sin \theta$$

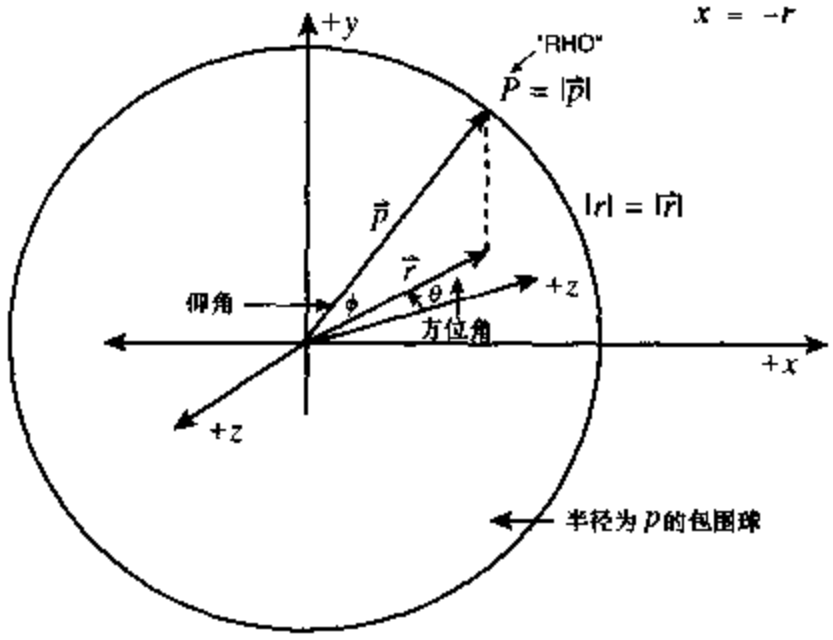


图 7.9　基于球面坐标系的 UVN 系统

假设相机位置的球面坐标 $\mathbf{p}(\rho, \Phi, \theta)$：

P = 目标向量。

ρ = 目标向量的长度，这里为 1.0，因为位于单位球体内。

θ = 方位角。

Φ = 仰角。

r = 目标向量在 x-z 平面上的投影长度。

则：

$r = \rho * \sin(\Phi)$

因此：

$x = -r * \sin(\theta)$

$y = \rho * \cos(\Phi)$

$z = r * \cos(\theta)$

从函数 SPHERICAL3D_To_POINT3D() 的结果可知，它是基于右手坐标系来生成 x、y 和 z 的。要将它们映射到现在的系统中，需要执行如下变换：

x = -y

y = z

z = x

这样，计算得到的 $\mathbf{p}(x, y, z)$ 正好是注视目标，这表明使用方位角、仰角和位置可以完整地定义 UVN 相机，因为使用这些值可以计算出其他的值。

至此，对 UVN 系统的介绍就结束了。使用欧拉系统和 UVN 系统，将能够创建出任何类型的相机。稍后将介绍设置这两种相机的代码以及生成相机变换矩阵的函数。

3. 初始化相机

正如前面指出的，相机函数不建立相机变换矩阵（这项工作由辅助函数完成），而是设置与视平面、裁剪面等相关的各项信息，如下所示：

```
void Init_CAM4DV1(CAM4DV1_PTR cam,    // 相机对象
          int cam_attr,      // 相机属性
```

```
          POINT4D_PTR cam_pos,  // 相机的初始位置
          VECTOR4D_PTR cam_dir, // 相机的初始角度
          POINT4D_PTR cam_target, // UVN 相机的初始目标位置
          float near_clip_z,   // 近裁剪面和远裁剪面
          float far_clip_z,
          float fov,       // 视野，单位为度
          float viewport_width, // 屏幕/视口大小
          float viewport_height)
{
// 这个函数初始化相机对象 cam
// 执行的错误检查不多
// 这里旨在让读者能够根据需要指定投影方式
// 同时尽可能地减少该函数所需的参数

cam->attr = cam_attr;       // 相机属性

VECTOR4D_COPY(&cam->pos, cam_pos); // 位置
VECTOR4D_COPY(&cam->dir, cam_dir); // 欧拉相机的方向向量或角度

// 对于 UVN 相机
VECTOR4D_INITXYZ(&cam->u, 1,0,0); // 设置为+x 轴方向
VECTOR4D_INITXYZ(&cam->v, 0,1,0); // 设置为+y 轴方向
VECTOR4D_INITXYZ(&cam->n, 0,0,1); // 设置为+z 轴方向

if (cam_target!=NULL)
  VECTOR4D_COPY(&cam->target, cam_target); // UVN 目标位置
else
  VECTOR4D_ZERO(&cam->target);

cam->near_clip_z = near_clip_z;   // 近裁剪面
cam->far_clip_z = far_clip_z;   // 远裁剪面

cam->viewport_width = viewport_width;  // 视口的大小
cam->viewport_height = viewport_height;

cam->viewport_center_x = (viewport_width-1)/2; // 视口的中心
cam->viewport_center_y = (viewport_height-1)/2;

cam->aspect_ratio = (float)viewport_width/(float)viewport_height;

// 将所有的矩阵都设置为单位矩阵
MAT_IDENTITY_4X4(&cam->mcam);
MAT_IDENTITY_4X4(&cam->mper);
MAT_IDENTITY_4X4(&cam->mscr);

// 设置相关变量
cam->fov      = fov;

// 将视平面大小设置为 2×(2/ar)
cam->viewplane_width = 2.0;
cam->viewplane_height = 2.0/cam->aspect_ratio;
// 根据 fov 和视平面大小计算视距
float tan_fov_div2 = tan(DEG_TO_RAD(fov/2));

cam->view_dist = (0.5)*(cam->viewplane_width)*tan_fov_div2;

// 判断 fov 是否为 90 度
if (fov == 90.0)
  {
   // 建立裁剪面
   POINT3D pt_origin; // 裁剪面上的一个点
   VECTOR3D_INITXYZ(&pt_origin,0,0,0);

   VECTOR3D vn; // 面法线

   // 右裁剪面
   VECTOR3D_INITXYZ(&vn,1,0,-1); // 平面 x=z
```

```
    PLANE3D_Init(&cam->rt_clip_plane, &pt_origin, &vn, 1);

    // 左裁剪面
    VECTOR3D_INITXYZ(&vn,-1,0,-1); // 平面-x=z
    PLANE3D_Init(&cam->lt_clip_plane, &pt_origin, &vn, 1);

    // 上裁剪面
    VECTOR3D_INITXYZ(&vn,0,1,-1); // 平面 y=z
    PLANE3D_Init(&cam->tp_clip_plane, &pt_origin, &vn, 1);

    // 下裁剪面
    VECTOR3D_INITXYZ(&vn,0,-1,-1); // 平面-y=z
    PLANE3D_Init(&cam->bt_clip_plane, &pt_origin, &vn, 1);
  }// end if d=1
else
  {
   // 计算 fov 不为 90 度时的裁剪面
   POINT3D pt_origin; // 平面上的一个点
   VECTOR3D_INITXYZ(&pt_origin,0,0,0);

   VECTOR3D vn; // 面法线

   // 由于 fov 不为 90 度，因此计算起来要复杂些
   // 解决这种问题的几何学方法很多，这里采用的方法是，
   // 首先计算表示裁剪面在平面 x-z 和 y-z 上的 2D 投影的向量
   // 然后计算与这两个向量垂直的向量，它就是裁剪面的法线
   // 右裁剪面，请在草稿纸上验证这种方法
   VECTOR3D_INITXYZ(&vn,cam->view_dist,0,-cam->viewplane_width/2.0);
   PLANE3D_Init(&cam->rt_clip_plane, &pt_origin, &vn, 1);

   // 左裁剪面，可以绕 Z 轴反射右裁剪面的法线，来得到左裁剪面的法线
   // 因为这两个裁剪面是关于 Z 轴对称的，因此只需对 x 求负
   VECTOR3D_INITXYZ(&vn,-cam->view_dist,0,-cam->viewplane_width/2.0);
   PLANE3D_Init(&cam->lt_clip_plane, &pt_origin, &vn, 1);

   // 上裁剪面
   VECTOR3D_INITXYZ(&vn,0,cam->view_dist,-cam->viewplane_width/2.0);
   PLANE3D_Init(&cam->tp_clip_plane, &pt_origin, &vn, 1);

   // 下裁剪面
   VECTOR3D_INITXYZ(&vn,0,-cam->view_dist,-cam->viewplane_width/2.0);
   PLANE3D_Init(&cam->bt_clip_plane, &pt_origin, &vn, 1);
  } // end else

} // end Init_CAM4DV1
```

请花些时间研究该函数。它一点也不复杂，但说明了相机和投影问题以及如何避开约束条件。

这个函数首先根据参数值设置相应的相机信息，然后根据视野和视口大小，将 x 作为主轴（其视野必须为指定的值）来计算其他信息。该函数现在并不关心使用的是哪种相机模型，它只是记录相机参数，并设置几个字段，创建相机矩阵 mcam 的工作是由辅助函数完成的。下面来看一些简单的例子。

范例 1：相机模型为欧拉相机，使用归一化相机坐标，视野为 90 度，视距为 1.0，屏幕大小为 400×400。该函数计算视平面大小，这里为 2×2。图 7.10 说明了这个相机模型。

```
// 初始化相机
Init_CAM4DV1(&cam,   // 相机对象
       CAM_MODEL_EULER, // 相机模型
       &cam_pos, // 相机的初始位置
       &cam_dir, // 相机的初始角度
       NULL,   // 不用于欧拉相机，因此设置为 NULL
       50.0,   // 远近裁剪面
       500.0,
       90.0,   // 视野，单位为度
       400,  // 屏幕/视口大小
       400);
```

范例 2：相机模型为 UVN 相机，视野为 90 度，屏幕大小为 640×480，进行简化的屏幕投影。该函数

计算视距。图 7.11 说明了这个相机模型。

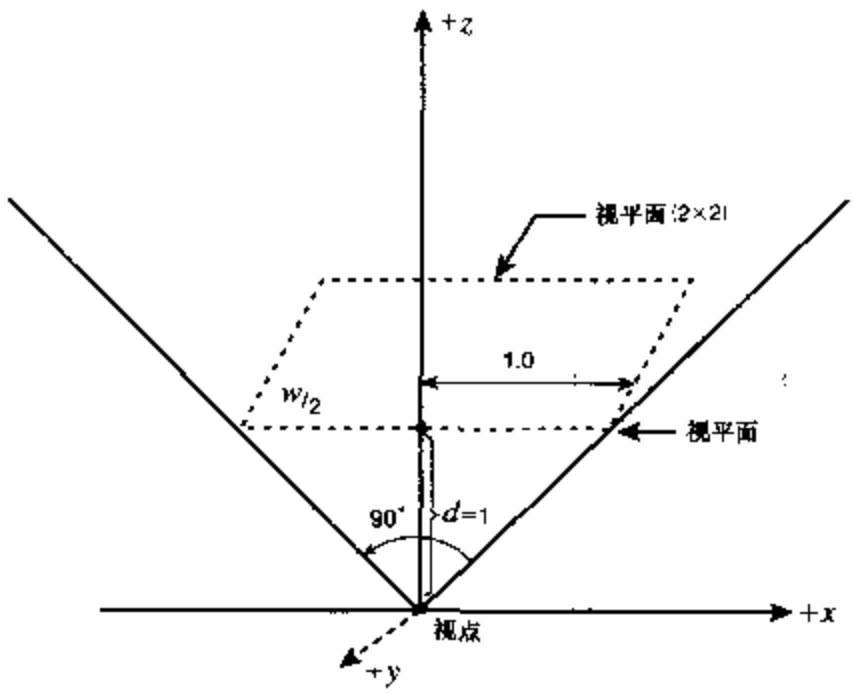


图 7.10　使用归一化坐标，视野为 90 度，视距为 1.0 的相机设置

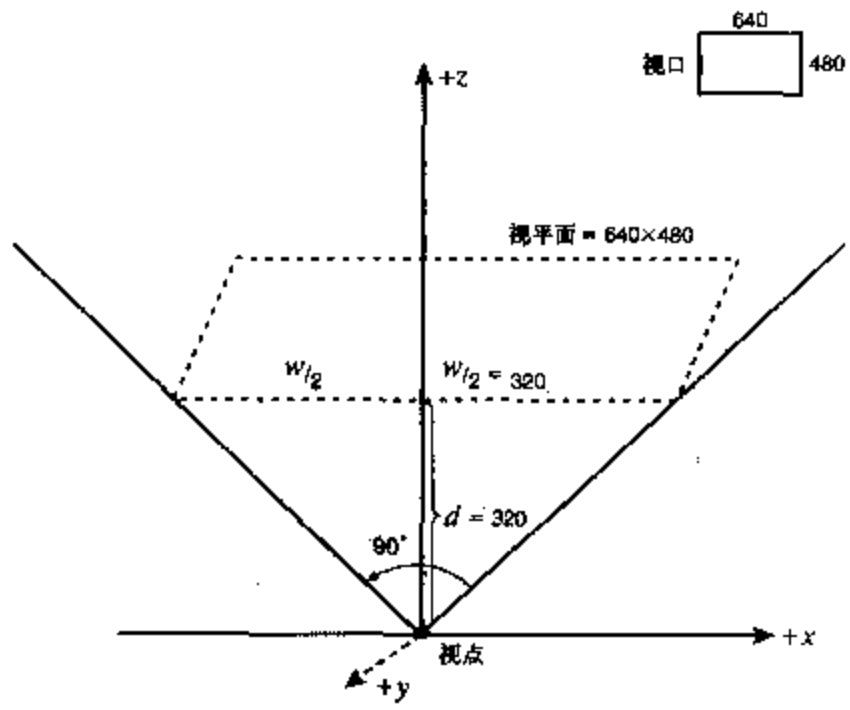


图 7.11　视野为 90 度，视平面大小与视口相同的相机设置

```
// 初始化相机
Init_CAM4DV1(&cam,   // 相机对象
       CAM_MODEL_UVN, // 相机模型
       &cam_pos, // 相机的初始位置
       &cam_dir, // 相机的初始角度
       &cam_target, // 初始目标位置
       50.0,   // 远近裁剪面
       500.0,
       90.0,   // 视野，单位为度
       640,  // 屏幕/视口大小
       480);
```

在这两个例子中，函数都计算了视平面、视距和宽高比。一般而言，范例 1 和 2 的相机设置足以满足我们的需要。我们总是将视野设置为 90 度（或 60 度），并采用归一化投影：视平面为 2×2（视口不是方形时，为 2×2/aspect_ratio）。

有关相机的工作几乎全部完成，余下的工作是指定相机位置和计算相机变换矩阵。下面的函数根据相机对象的位置和欧拉角度，计算各种矩阵及其他元素：

```
void Build_CAM4DV1_Matrix_Euler(CAM4DV1_PTR cam, int cam_rot_seq)
{
// 这个函数根据欧拉角度计算相机变换矩阵
// 并将其存储到传入的相机对象中
// 第 6 章介绍过，要创建相机变换矩阵，需要这样做：

// Mcam = mt(-1) * my(-1) * mx(-1) * mz(-1)
// 即相机平移矩阵的逆矩阵乘以相机绕 y、x、z 轴的旋转矩阵的逆矩阵
// 采用什么样的旋转顺序完全取决于用户，因此这里没有强制采用某种顺序
// 而是根据参数 cam_rot_seq 的值来决定采用哪种顺序
// 该参数的取值为 CAM_ROT_SEQ_XYZ，其中 XYZ 可以按任何顺序排列，即 YXZ、ZXY 等

MATRIX4X4 mt_inv, // 相机平移矩阵的逆矩阵
     mx_inv, // 相机绕 x 轴的旋转矩阵的逆矩阵
     my_inv, // 相机绕 y 轴的旋转矩阵的逆矩阵
     mz_inv, // 相机绕 z 轴的旋转矩阵的逆矩阵
     mrot,  // 所有逆旋转矩阵的积
     mtmp;  // 用于存储临时矩阵

// 第 1 步：根据相机位置计算相机平移矩阵的逆矩阵
Mat_Init_4X4(&mt_inv, 1,  0,   0,   0,
           0,  1,   0,   0,
           0,  0,   1,   0,
           -cam->pos.x, -cam->pos.y, -cam->pos.z, 1);

// 第 2 步：创建旋转矩阵的逆矩阵
```

```
// 要计算正规旋转矩阵的逆矩阵，可以将其转置，
// 也可以将每个旋转角度取负

// 首先计算 3 个旋转矩阵的逆矩阵

// 提取欧拉角度
float theta_x = cam->dir.x;
float theta_y = cam->dir.y;
float theta_z = cam->dir.z;

// 计算角度 x 的正弦和余弦
float cos_theta = Fast_Cos(theta_x); // cos(-x) = cos(x)，因此余弦值不变
float sin_theta = -Fast_Sin(theta_x); // sin(-x) = -sin(x)

// 建立矩阵
Mat_Init_4X4(&mx_inv, 1,  0,     0,     0,
             0,   cos_theta, sin_theta, 0,
             0,   -sin_theta, cos_theta, 0,
             0,   0,    0,    1);

// 计算角度 y 的正弦和余弦
cos_theta = Fast_Cos(theta_y); // cos(-x) = cos(x)，因此余弦值不变
sin_theta = -Fast_Sin(theta_y); // sin(-x) = -sin(x)
// 建立矩阵
Mat_Init_4X4(&my_inv,cos_theta, 0, -sin_theta, 0,
             0,     1, 0,           0,
             sin_ theta, 0, cos_theta, 0,
             0,     0, 0,           1);

// 计算角度 z 的正弦和余弦
cos_theta = Fast_Cos(theta_z); // cos(-x) = cos(x)，因此余弦值不变
sin_theta = -Fast_Sin(theta_z); // sin(-x) = -sin(x)

// 建立矩阵
Mat_Init_4X4(&mz_inv, cos_theta, sin_theta, 0, 0,
             -sin_theta, cos_theta, 0, 0,
             0,     0,           1, 0,
             0,     0,           0, 1);

// 现在计算逆旋转矩阵的乘积
switch(cam_rot_seq)
    {
    case CAM_ROT_SEQ_XYZ:
    {
    Mat_Mul_4X4(&mx_inv, &my_inv, &mtmp);
    Mat_Mul_4X4(&mtmp, &mz_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_YXZ:
    {
    Mat_Mul_4X4(&my_inv, &mx_inv, &mtmp);
    Mat_Mul_4X4(&mtmp, &mz_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_XZY:
    {
    Mat_Mul_4X4(&mx_inv, &mz_inv, &mtmp);
    Mat_Mul_4X4(&mtmp, &my_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_YZX:
    {
    Mat_Mul_4X4(&my_inv, &mz_inv, &mtmp);
    Mat_Mul_4X4(&mtmp, &mx_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_ZYX:
```

```
    {
    Mat_Mul_4X4(&mz_inv, &my_inv, &mtmp);
    Mat_Mul_4X4(&mtmp, &mx_inv, &mrot);
    } break;

    case CAM_ROT_SEQ_ZXY:
    {
    Mat_Mul_4X4(&mz_inv, &mx_inv, &mtmp);
    Mat_Mul_4X4(&mtmp, &my_inv, &mrot);

    } break;

    default: break;
    } // end switch

// 现在，mrot 逆旋转矩阵的乘积
// 接下来将其乘以逆平移矩阵，并将结果存储到相机对象的相机变换矩阵中
Mat_Mul_4X4(&mt_inv, &mrot, &cam->mcam);

} // end Build_CAM4DV1_Matrix_Euler
```

这个函数接收两个参数：相机对象和指定欧拉角度旋转顺序的标记。欧拉角度和相机位置存储在相机对象中，因此只需要调用该函数之前，在相机对象中设置它们。旋转顺序的可能取值如下：

```
// 定义相机旋转顺序对应的值
#define CAM_ROT_SEQ_XYZ 0
#define CAM_ROT_SEQ_YXZ 1
#define CAM_ROT_SEQ_XZY 2
#define CAM_ROT_SEQ_YZX 3
#define CAM_ROT_SEQ_ZYX 4
#define CAM_ROT_SEQ_ZXY 5
```

假设已经创建了一个名为 cam 的相机对象，并调用 Init_CAM4DV1()对其进行了初始化。要命令相机对象按旋转顺序 ZYX 来创建矩阵，可以这样做：

```
// 设置相机位置
cam.pos.x = 100;
cam.pos.y = 200;
cam.pos.z = 300;
// 设置观察角度（单位为度）
cam.dir.x = -45;
cam.dir.y = 0;
cam.dir.z = 0;

// 生成相机变换矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
```

然后，便可以使用矩阵 cam.mcam 来执行世界坐标到相机坐标变换。图 7.12 显示了采用上述设置的虚

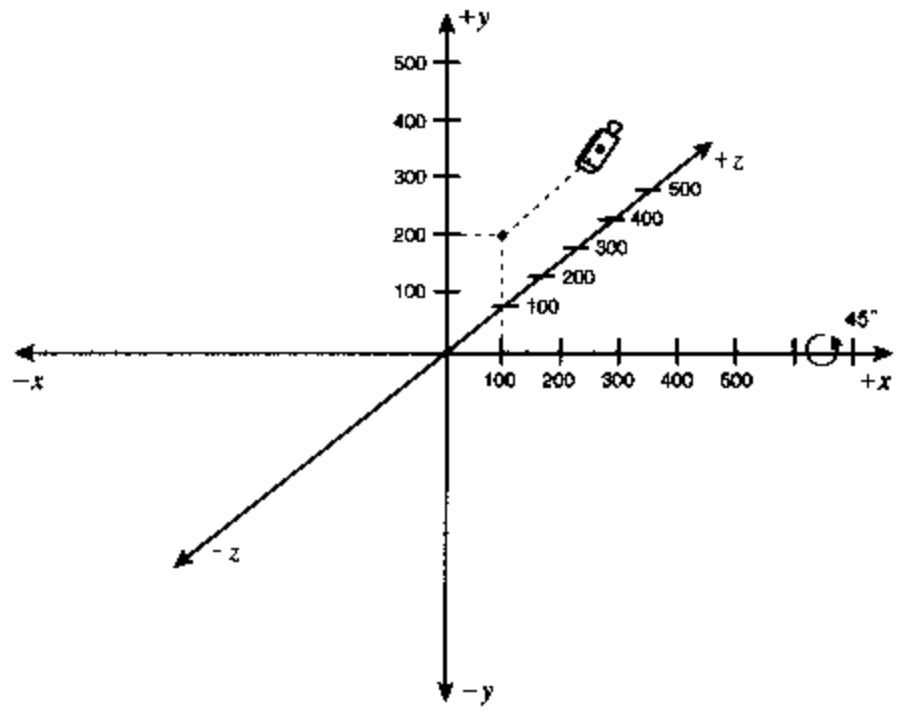


图 7.12　调用相机设置函数的结果

拟相机的位置和朝向。然而，矩阵 cam.mper 和 cam.mscr 还未定义。

UVN 模型的相机函数如下：

```
void Build_CAM4DV1_Matrix_UVN(CAM4DV1_PTR cam, int mode)
{
// 这个函数根据注视向量 n、上向量 v 和右向量 u 创建一个相机变换矩阵
// 并将其存储到传入的相机对象中，这些值都是从相机对象中提取的
// 参数 mode 指定如何计算 uvn
// UVN_MODE_SIMPLE：低级简单模型，使用目标位置和观察参考点；
// UVN_MODE_SPHERICAL：球面坐标模式，分量 x 和 y 被用作观察向量的方位角和仰角，
// 观察参考点为相机位置。
MATRIX4X4 mt_inv, // 逆相机平移矩阵
     mt_uvn,  // UVN 相机变换矩阵
     mtmp;   // 用于存储临时矩阵

//第 1 步：根据相机位置创建逆平移矩阵
Mat_Init_4X4(&mt_inv, 1,  0,   0,   0,
           0,  1,  0,   0,
           0,  0,  1,   0,
           -cam->pos.x, -cam->pos.y, -cam->pos.z, 1);


// 第 2 步：确定如何计算目标点
if (mode == UVN_MODE_SPHERICAL)
   {
   // 使用球面坐标模式
   // 需要重新计算目标点

   // 提取方位角和仰角
   float phi   = cam->dir.x; // 仰角
   float theta = cam->dir.y; // 方位角

   // 计算三角函数
   float sin_phi = Fast_Sin(phi);
   float cos_phi = Fast_Cos(phi);

   float sin_theta = Fast_Sin(theta);
   float cos_theta = Fast_Cos(theta);

   // 计算目标点在单位球面上的位置（x、y 和 z）
   cam->target.x = -1*sin_phi*sin_theta;
   cam->target.y = 1*cos_phi;
   cam->target.z = 1*sin_phi*cos_theta;
   } // end else
// 至此，有了为重新计算 U、V 和 N 所需的全部参数：观察参考点和目标点
// 第 1 步：n = <目标位置 - 观察参考点>
VECTOR4D_Build(&cam->pos, &cam->target, &cam->n);

// 第 2 步：将 v 设置为<0,1,0>
VECTOR4D_INITXYZ(&cam->v,0,1,0);

// 第 3 步：u = (v x n)
VECTOR4D_Cross(&cam->v,&cam->n,&cam->u);

// 第 4 步：v = (n x u)
VECTOR4D_Cross(&cam->n,&cam->u,&cam->v);

// 第 5 步：对所有向量都进行归一化
VECTOR4D_Normalize(&cam->u);
VECTOR4D_Normalize(&cam->v);
VECTOR4D_Normalize(&cam->n);


// 将 u、v、n 代入，得到 UVN 旋转矩阵
Mat_Init_4X4(&mt_uvn, cam->u.x,  cam->v.x,   cam->n.x,   0,
           cam->u.y,  cam->v.y,  cam->n.y,   0,
           cam->u.z,  cam->v.z,  cam->n.z,   0,
```

```
            0,      0,      0,      1);

// 将平移矩阵乘以 uvn 矩阵，并将结果存储到相机变换矩阵 mcam 中
Mat_Mul_4X4(&mt_inv, &mt_uvn, &cam->mcam);

} // end Build_CAM4DV1_Matrix_UVN
```

现在可以回到前面暂停的 3D 流水线了。

7.3.5　世界坐标到相机坐标变换

建立相机模型并对其进行设置后，便可以执行世界坐标到相机坐标变换，但在此之前先来简要回顾一下相机创建函数为何管用。UVN 模型和欧拉模型根据相机的位置和朝向信息创建一个矩阵，这包括两步：平移和旋转。

图 7.13 说明了世界坐标到相机坐标变换的这两个步骤。在该图中，通过一台虚拟相机来观察 3D 世界，这台相机位于某个位置，具有某种朝向以及特定的水平视野和垂直视野。我们关心的是，如果世界和相机都是真的，通过相机镜头将看到什么样的风景。为此，需要创建一个变换矩阵，用于对世界中的所有物体进行变换，使得相机就像位于原点(0，0，0)，并朝向+Z 轴方向。这样，可以简化流水线后面的透视投影的数学计算。

前面介绍过，可以这样来完成这项工作：假设相机位置的世界坐标为 cam_pos(cam_x，cam_y，cam_z)，这个位置是由矩阵 $\mathbf{T}_{cam}$ 定义的，则第一步是使用该矩阵的逆矩阵（-cam_pos=(-cam_x，-cam_y，-cam_z)），即 $\mathbf{T}_{cam}^{-1}$ 对世界中的所有物体、顶点和多边形等进行平移。该矩阵如下：

$$\mathbf{T}_{cam}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -cam\_x & -cam\_y & -cam\_z & 1 \end{bmatrix}$$

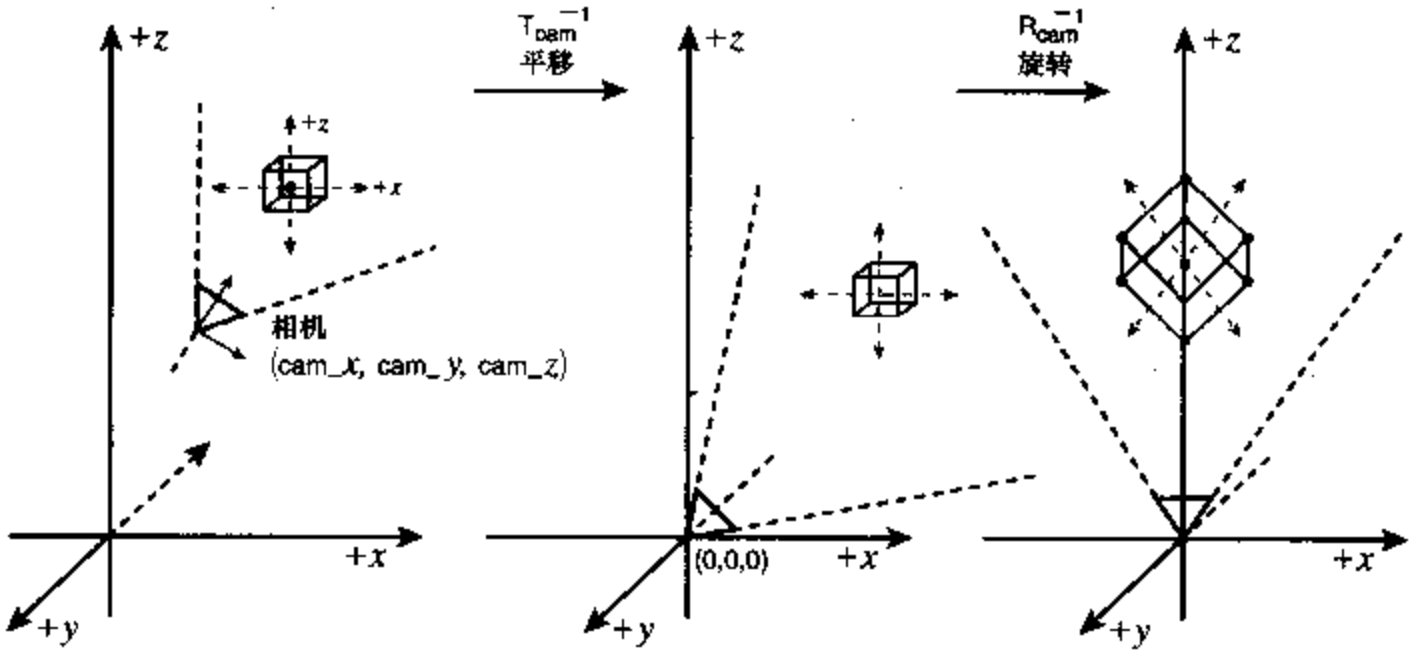


图 7.13　世界坐标到相机坐标变换

使用这个矩阵对每个顶点进行变换后，所有几何体都将被平移到相机位于原点时的相对位置。相机变换的下一步是旋转，这一步要复杂些，表示相机朝向的 4×4 矩阵随相机模型而异。在欧拉相机模型中，使用 3 个角度来表示相机的朝向：cam_ang = (ang_x，ang_y，ang_z)，如图 7.14 所示。各个角度分别表示绕各个正轴顺时针或逆时针（右手坐标系）旋转的程度，这些变换分别被称为 $\mathbf{R}_{camy}$、$\mathbf{R}_{camx}$ 和 $\mathbf{R}_{camz}$。应用这些变换的顺序（XYZ、YZX 等）至关重要，在大多数情况下，我们将使用顺序 YXZ 和 ZYX。

因此，要执行相机变换的旋转步骤，需要按如下顺序应用这些旋转变换的逆变换：

$\mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1} * \mathbf{R}_{camz}^{-1}$

或

$\mathbf{R}_{camz}^{-1} * \mathbf{R}_{camy}^{-1} * \mathbf{R}_{camx}^{-1}$

将平移和旋转组合起来，相机变换 $\mathbf{T}_{wc}$ 如下：

$$T_{wc} = T_{cam}^{-1} * R_{camy}^{-1} * R_{camx}^{-1} * R_{camz}^{-1}$$

或：

$$T_{wc} = T_{cam}^{-1} * R_{camz}^{-1} * R_{camy}^{-1} * R_{camx}^{-1}$$

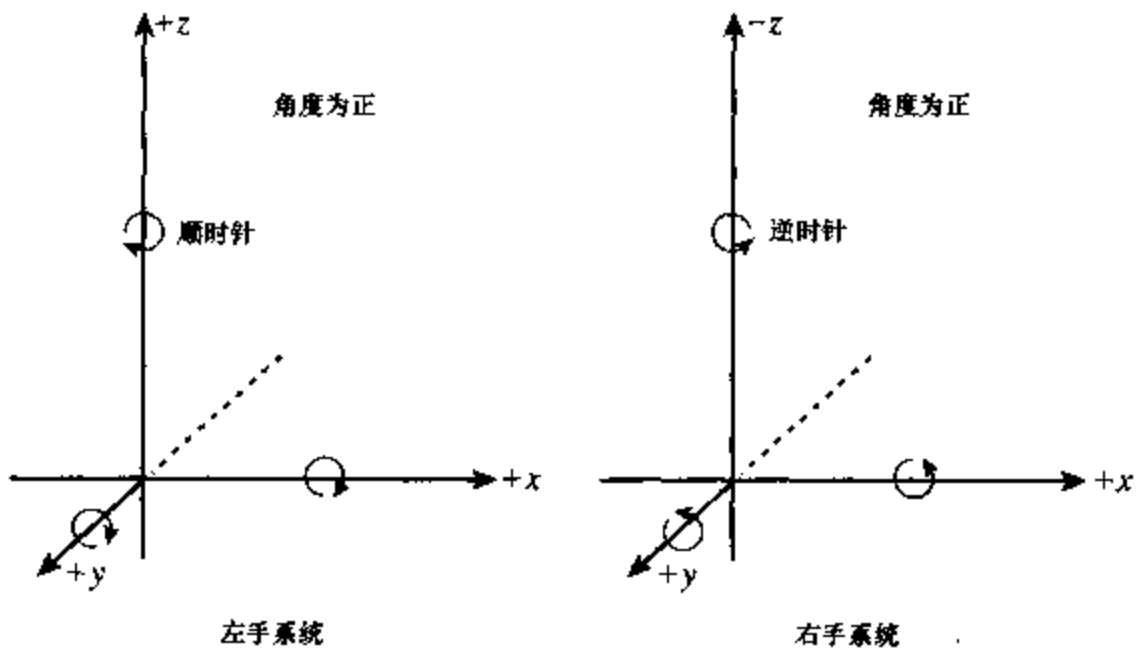


图 7.14　相机旋转角度和坐标系之间的关系

无论采用哪种方式，都需要先执行一次平移，然后执行 3 次旋转。当然，可以将这些操作合并成一个名为 T_{wc} 的矩阵，如前面所示。为执行世界坐标到相机坐标变换，只需要使用矩阵 T_{wc} 对每个顶点进行变换即可。我们有一个创建和初始化相机的函数以及计算世界坐标到相机坐标变换矩阵的函数 Build_CAM4DV1_Matrix_Euler()。函数 Build_CAM4DV1_Matrix_UVN()的功能与此相同，但使用的方法不同，即它使用上向量、右向量和目标向量，而不是角度，但这两个函数都计算需要将其应用于每个顶点的变换矩阵。

因此，我们需要的只是一个将变换应用于物体和/或渲染列表的函数。我们已经有两个通用变换函数：void Transform_OBJECT4DV1()和 void Transform_RENDERLIST4DV1()，它们可接受任何矩阵作为参数，并对物体或渲染列表进行变换。可以使用这两个函数来完成这项任务，但作者喜欢编写专用函数（有时候为流水线中的每个步骤编写专用函数）而不是使用通用函数，以便能够充分进行优化。下面介绍这两种方法。

1. 物体的世界坐标到相机坐标变换

正如前面指出的，通过调用函数 Transform_OBJECT4DV1()，可以使用任何 4×4 矩阵对 OBJECT4DV1 进行变换，我们首先介绍这种方法。假设有一个名为 obj 的物体和一个名为 cam 的相机对象，且所有的初始化工作都已完成，则可以这样做：

```
// 生成欧拉相机变换矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// 应用变换矩阵：对世界坐标进行变换
Transform_OBJECT4DV1(&obj, &cam.mcam, TRANSFORM_TRANS_ONLY, 1 );
```

这样，物体将位于相机空间中，可以对其进行透视投影和屏幕映射。另一方面，也可以编写一个专用函数，而不使用函数 Transform_OBJECT4DV1()，其代码如下：

```
void World_To_Camera_OBJECT4DV1(CAM4DV1_PTR cam, OBJECT4DV1_PTR obj)
{
// 这是一个基于矩阵的函数
// 它根据传入的相机变换矩阵，将物体的世界坐标变换为相机坐标
// 它完全不考虑多边形本身，只是对 vlist_trans[]中的顶点进行变换
// 这是变换方法之一，您可能选择对渲染列表进行变换
// 因为渲染列表中的多边形表示的几何体都通过了背面剔除

// 将物体的每个顶点变换为相机坐标
// 这里假设顶点已经被变换为世界坐标
// 且结果存储在 vlist_trans[]中
```

```
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
   {
   // 使用相机对象中的矩阵 mcam 对顶点进行变换
   POINT4D presult; // 用于存储每次变换的结果

   // 对顶点进行变换
   Mat_Mul_VECTOR4D_4X4(&obj->vlist_trans[vertex],&cam->mcam, presult);

   // 将结果存回去
   VECTOR4D_COPY(&obj->vlist_trans[vertex], &presult);
   } // end for vertex

} // end World_To_Camera_OBJECT4DV1
```

这个函数的速度更快，也更紧凑。它接受一个物体和一台相机作为参数。设置好物体和相机后，可以按如下方式来调用该函数：

```
World_To_Camera_OBJECT4DV1(&cam, &obj);
```

注意：在实际的 3D 软件引擎中，作者可能使用硬编码来执行矩阵和向量运算，一般而言，在整个引擎中都这样做可将速度提高 5% ~ 10%。如果需要提高速度，这种方法很不错；否则，没必要使用这种方法而导致代码混乱。在设计阶段，最好设计清晰的算法，而不要对代码进行优化。

2. 渲染列表的世界坐标到相机坐标变换

对于渲染列表，执行世界坐标到相机坐标变换的方式与物体相同。可以使用通用的变换函数，也可以编写专用函数来对渲染列表进行变换。同样，这里也将介绍这两种方法，首先介绍使用通用函数的方法：

```
// 生成相机变换矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// 应用变换：对世界坐标进行变换
Transform_RENDERLIST4DV1(&rend_list, &cam.mcam, TRANSFORM_TRANS_ONLY);
```

函数 Transform_RENDERLIST4DV1()没有参数 transform_basis，这是因为多边形列表表示很多物体，同时从物体中提取多边形并将其插入多边形列表中后，这些物体的朝向便无关紧要。下面是执行渲染列表变换的专用函数。惟一复杂的地方是，需要遍历列表中的每个多边形；而在对 OBJECT4DV1 进行变换的专用函数中，只需要对一个顶点列表进行变换。

```
void World_To_Camera_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                  CAM4DV1_PTR cam)
{
// 这个函数是基于矩阵的
// 它根据传入的相机变换矩阵将渲染列表中每个多边形变换为相机坐标
// 如果在流水线的上游已经将每个物体转换为多边形
// 并将它们插入到渲染列表中
// 将使用这个函数，而不是基于物体的函数对顶点进行变换
// 将物体转换为多边形的操作是在物体剔除、局部变换、局部坐标到世界坐标变换
// 以及背面消除之后进行的
// 这样最大限度地减少了每个物体中被插入到渲染列表中的多变形数目
// 这个函数假设至少已经进行了局部坐标到世界坐标变换
// 且多边形数据存储在 POLYF4DV1 的变换后的列表 tvlist 中

// 将渲染列表中的每个多边形变换为相机坐标
// 假设每个多边形都已变换为世界坐标，且结果存储在 tvlist[]中

for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// 获取下一个多边形
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// 这个多边形是否有效?
// 当且仅当多边形处于活动状态、可见时才对它进行变换
// 但对于线框引擎，背面的概念无关紧要
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
```

```
    (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形
// 满足条件，对其进行变换
for (int vertex = 0; vertex < 3; vertex++)
   {
   // 使用相机对象中的矩阵 mcam 对顶点进行变换
   POINT4D presult; // 用于存储每次变换的结果

   // 对顶点进行变换
   Mat_Mul_VECTOR4D_4X4(&curr_poly->tvlist[vertex],&cam->mcam,&presult);

   // 将结果存回去
   VECTOR4D_COPY(&curr_poly->tvlist[vertex], &presult);
   } // end for vertex

} // end for poly

} // end World_To_Camera_RENDERLIST4DV1
```

要对渲染列表 rend_list 进行变换，可以这样调用上述函数（这里假设 cam 已被定义和初始化）：

```
World_To_Camera_RENDERLIST4DV1(&cam, &rend_list);
```

至此，流水线中的步骤已完成了一半左右。

现在，需要完成透视投影变换和屏幕变换。然而，这里需要指出的是，我们还未讨论过裁剪，这可是一个大主题。然而，前一章指出过，裁剪可以在 3D 物体空间中进行，也可以在 2D 光栅图像空间中进行。

然而，绝对不能等到光栅化阶段才进行裁剪，因为无法对 z 值为负（更糟的是为 0.0，将导致除以 0.0 的错误）的多边形顶点进行投影。这一点必须牢记。必须采用某种方式对裁剪进行处理，但在此之前先来讨论一下将整个物体从流水线中删除的问题。

7.3.6 物体剔除

物体剔除（culling）指的是将物体删除，以免在整个流水线中对其进行处理，如图 7.15 所示。物体剔除可以在世界空间中进行，也可以在相机空间中进行，作者喜欢采用后一种方式。但读者可能会问，如果到相机空间才去剔除物体，对于那些在相机空间不可见的物体，之前将其从世界坐标变换为相机坐标不是浪费 CPU 周期吗？答案是，不一定。

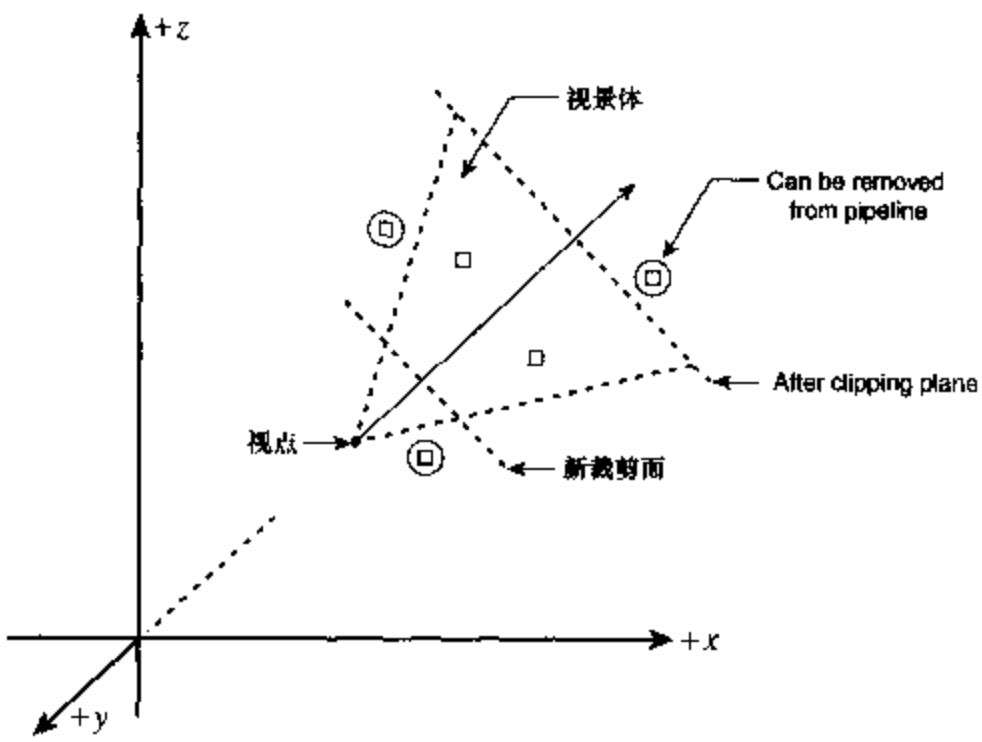


图 7.15 将物体从 3D 流水线中删除

如果使用物体的中心和最大半径来创建包围球，为执行物体剔除测试，可以将表示物体中心的单个点从世界坐标变换为相机坐标，然后判断包围球是否在视景体内。前一章对此做了详细介绍，最后的数学公

式只是使用左、右、上、下、远、近裁剪面对几个点进行测试。无论如何，这是 3D 引擎流水线中最重要的步骤，因为在大多数情况下，只有几个物体位于场景中，其他物体要么在视野范围外，要么在观察者后面，因此没有理由对它们进行处理。所以，引擎流水线中绝对不能没有这样的功能。

要实现物体剔除，需要编写一个函数，它接受物体的世界空间位置和平均/最大半径作为参数，并根据当前的相机变换来剔除物体。这种方法只适用于物体，因为将物体变换为多边形并将其插入到主渲染列表中后，将不再有物体的概念。物体消除函数如下：

```
int Cull_OBJECT4DV1(OBJECT4DV1_PTR obj, // 要对其执行剔除操作的物体
          CAM4DV1_PTR cam,  // 剔除时使用的相机
          int cull_flags)   // 要考虑的裁剪面
{
// 这个函数是基于矩阵的
// 它根据传入的相机信息判断物体是否在视景体内
// 参数 cull_flags 指定在哪些轴上执行剔除
// 其取值为各种剔除标记的 OR
// 如果物体被剔除，将相应地设置其状态
// 这个函数假设相机和物体都是有效的

// 第 1 步：将物体的包围球球心变换为相机坐标

POINT4D sphere_pos; // 用于存储包围球球心变换后的坐标

// 对点进行变换
Mat_Mul_VECTOR4D_4X4(&obj->world_pos, &cam->mcam, &sphere_pos);

// 第 2 步：根据剔除标记对物体执行剔除操作
if (cull_flags & CULL_OBJECT_Z_PLANE)
{
// 只根据远近裁剪面来剔除物体

// 使用远近裁剪面进行测试
if ( ((sphere_pos.z - obj->max_radius) > cam->far_clip_z) ||
  ((sphere_pos.z + obj->max_radius) < cam->near_clip_z) )
  {
  SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
  return(1);
  } // end if

} // end if

if (cull_flags & CULL_OBJECT_X_PLANE)
{
// 只根据左右裁剪面进行物体剔除
// 本可以使用平面方程，但使用三角形相似更容易
// 因为这是一种 2D 问题
// 如果视野为 90 度，问题将更简单
// 这里假设视野不为 90 度

// 使用右裁剪面和左裁剪面检测包围球上最左边和最右边的点
float z_test = (0.5)*cam->viewplane_width*sphere_pos.z/cam->view_dist;

if ( ((sphere_pos.x-obj->max_radius) > z_test) || // 右边
   ((sphere_pos.x+obj->max_radius) < -z_test) ) // 左边
  {
  SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
  return(1);
  } // end if
} // end if

if (cull_flags & CULL_OBJECT_Y_PLANE)
{
// 只根据上下裁剪面对物体进行剔除
// 本可以使用平面方程，但使用三角形相似更容易
// 因为这是一种 2D 问题
// 如果视野为 90 度，问题将更简单
```

```
// 这里假设视野不为 90 度

// 使用上裁剪面和下裁剪面检测包围球上最下边和最上边的点
float z_test = (0.5)*cam->viewplane_height*sphere_pos.z/cam->view_dist;

if ( ((sphere_pos.y-obj->max_radius) > z_test) || // top side
  ((sphere_pos.y+obj->max_radius) < -z_test) ) // bottom side
  {
  SET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
  return(1);
  } // end if

} // end if

// return failure to cull
return(0);

} // end Cull_OBJECT4DV1
```

该函数很简单，它以数学方式实现了前一章得出的有关物体剔除的结论，这里不再重复。该函数接受要进行剔除测试的物体、当前相机（调用该函数之前，必须初始化矩阵 mcam）和一个标记（cull_flags，它指定要进行哪些测试）作为参数。

参数 cull_flags 指定要在哪些轴上进行剔除测试。例如，只在 z 轴上（根据近裁剪面和远裁剪面）进行剔除测试，物体的 x 坐标和 y 坐标仍可能使其不可见，但这样至少可以确保不会对位于观察者后面的物体进行投影。一般而言，应根据 3D 视景体的所有 6 个裁剪面对物体进行剔除，因为这种测试的开销非常低。剔除测试标记如下：

```
// 剔除标记
#define CULL_OBJECT_X_PLANE    0x0001 // 根据左右裁剪面进行剔除
#define CULL_OBJECT_Y_PLANE    0x0002 // 根据上下裁剪面进行剔除
#define CULL_OBJECT_Z_PLANE    0x0004 // 根据远近裁剪面进行剔除
#define CULL_OBJECT_XYZ_PLANES  (CULL_OBJECT_X_PLANE | CULL_OBJECT_Y_PLANE
                               | CULL_OBJECT_Z_PLANE)
```

要进行多种测试，只需使用 OR 运算将相应的标记组合起来即可。例如，要根据整个视景体来剔除物体，可以这样调用上述函数：

```
if (Cull_OBJECT4DV1(&obj, &cam,
            CULL_OBJECT_X_PLANE |
            CULL_OBJECT_Y_PLANE |
            CULL_OBJECT_Z_PLANE))
{ /* 物体被剔除 */ }
```

执行剔除后，该函数将修改物体的状态标记，对于被剔除的物体，设置相应的位以指出这一点。为简化这项工作，作者定义了一些用于表示 OBJECT4DV1 状态的常量：

```
// 物体的状态
#define OBJECT4DV1_STATE_ACTIVE      0x0001
#define OBJECT4DV1_STATE_VISIBLE     0x0002
#define OBJECT4DV1_STATE_CULLED      0x0004
```

后面将使用这些状态常量，而现在通过在渲染函数和变换函数中设置和检测 OBJECT4DV1_STATE_CULLED，可以不考虑被剔除的物体。另外，物体在某一帧中可能被剔除，而在另一帧中没有被剔除，因此，在每个游戏帧中都必须重置物体的标记，以清除所有的临时标记（如剔除标记）。下面是一个完成这项工作的简单函数：

```
void Reset_OBJECT4DV1(OBJECT4DV1_PTR obj)
{
// 这个函数重置传入的物体的状态，为变换做准备
// 通常是重置被剔除、被裁剪掉和背面等标记，也可以在这里做其他准备工作
// 物体是有效的，接下来重置其各个多边形的状态标记
// 重置物体的被剔除标记
RESET_BIT(obj->state, OBJECT4DV1_STATE_CULLED);
```

```
// 重置多边形的被裁剪掉和背面标记
for (int poly = 0; poly < obj->num_polys; poly++)
  {
  // 获得下一个多边形
  POLY4DV1_PTR curr_poly = &obj->plist[poly];

  // 首先判断多边形是否可见?
  if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE))
    continue; // 进入下一个多边形

  // 重置被裁剪掉和背面标记
  RESET_BIT(curr_poly->state, POLY4DV1_STATE_CLIPPED);
  RESET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);

  } // end for poly

} // end Reset_OBJECT4DV1
```

调用上述函数将重置物体的剔除标记以及组成该物体的每个多边形的背面标记。接下来讨论背面消除。

7.3.7　背面消除

背面消除指的是删除背向视点的多边形，如图 7.16 所示。前一章讨论了有关这方面的数学知识，这里不打算详细介绍。简单地说，将对物体或渲染列表的每个多边形进行测试。将计算一个从多边形到视点的方向向量和多边形外向法线以及它们之间的夹角。如果夹角大于 90 度（即 $n.l <= 0$），表明多边形为背面，是不可见的。

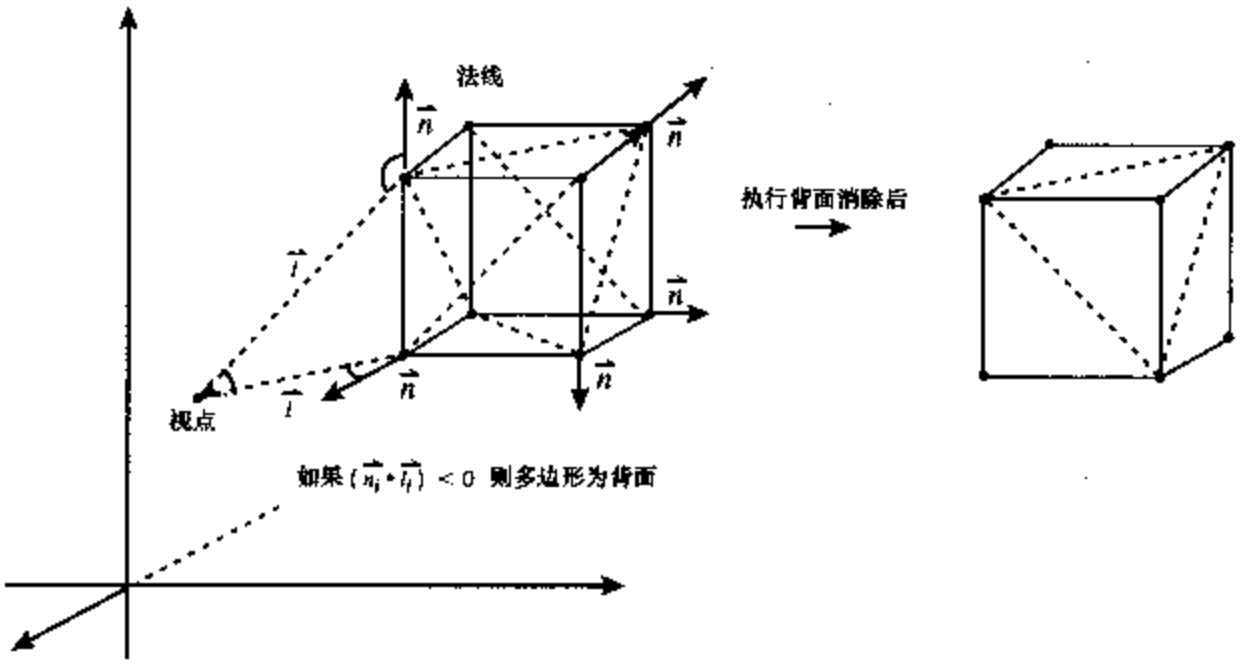


图 7.16　背面消除

当然，背面的概念只对单面多边形有意义。对于那些从两边都可以看到的（即双面）多边形，这种测试毫无意义。这种测试通常是在世界空间而不是相机空间进行的（因为只需要知道视点），这样可通过背面消除删除大量的多边形，避免对它们进行世界坐标到相机坐标变换。这是一种不错的删除一半几何体的方式。

1. 物体的背面消除

下面的函数删除结构 OBJECT4DV1 表示的物体的背面：

```
void Remove_Backfaces_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据数组 vlist_trans 中的顶点数据以及相机位置
// 消除物体的背面多边形
// 这里只设置多边形的背面状态

// 检查物体是否已被剔除
if (obj->state & OBJECT4DV1_STATE_CULLED)
  return;
```

```
// 处理物体的每个多边形
for (int poly=0; poly < obj->num_polys; poly++)
  {
  // 获取多边形
  POLY4DV1_PTR curr_poly = &obj->plist[poly];
// 该多边形是否有效?
// 判断多边形是否没有被裁剪掉、没有被剔除、处于活动状态、可见且不是双面的
if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
   (curr_poly->attr & POLY4DV1_ATTR_2SIDED)  ||
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
  continue; // 进入下一个多边形

// 获取顶点列表中的顶点索引
// 多变形不是自包含的，而是基于物体的顶点列表
int vindex_0 = curr_poly->vert[0];
int vindex_1 = curr_poly->vert[1];
int vindex_2 = curr_poly->vert[2];

// 我们将使用变换后的多边形顶点列表
// 因为背面消除只能在顶点被转换为世界坐标之后才能进行

// 需要计算多边形的面法线
// 顶点是按顺时针方向排列的，u = p0->p1, v=p0->p2, n=uxv
VECTOR4D u, v, n;

// 计算 u 和 v
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
       &obj->vlist_trans[ vindex_1 ], &u);
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
       &obj->vlist_trans[ vindex_2 ], &v);

// 计算叉积
VECTOR4D_Cross(&u, &v, &n);

// 创建指向视点的向量
VECTOR4D view;
VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ], &cam->pos, &view);

// 计算点积
float dp = VECTOR4D_Dot(&n, &view);
   // 如果 dp< 0，则多边形不可见
   if (dp <= 0.0 )
     SET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);

   } // end for poly

} // end Remove_Backfaces_OBJECT4DV1
```

请读者花几分钟阅读该函数，看它是否合理。这个函数接受一个物体作为参数，并遍历多边形列表，检查各个多边形是否处于活动状态且是单面的。如果是，则判断从相机对象定义的视点观察，它是否可见。在这个函数中，不需要矩阵 mcam，因为只需要知道相机的位置。要调用该函数，可以这样做：

```
Remove_Backfaces_OBJECT4DV1(&obj,&cam);
```

这样，在物体的多边形列表中，所有属于背面的多边形的标记都将被修改。具体地说，将像这样修改每个属于背面的多边形的背面标记位：

```
SET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);
```

提示：务必在执行物体剔除之后和世界坐标到相机坐标变换之前进行背面消除。

2. 渲染列表的背面消除

如果在背面消除之前物体被存储在一起，或者只有游戏空间的多边形列表而没有物体，则需要对渲染列表执行背面消除。当然，如果在将组成物体的多边形插入渲染列表时执行了背面消除，则没有必要对渲

染列表执行背面消除。对 RENDERLIST4DV1 执行背面消除的函数与用于 OBJECT4DV1 的函数几乎相同，只是访问数据结构的方式不同而已。该函数如下：

```
void Remove_Backfaces_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list, CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据多边形列表数据 tvlist 和相机位置
// 来消除多边形列表中属于背面的多边形
// 这里只设置多边形的背面状态
for (int poly = 0; poly < rend_list->num_polys; poly++)
  {
  // 获取当前多边形
  POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

  // 该多边形是否有效?
  // 判断多边形是否没有被裁剪掉、没有被剔除、处于活动状态、可见且不是双面的
  if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE)
    || (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
    (curr_poly->attr & POLY4DV1_ATTR_2SIDED)  ||
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // move onto next poly

    // 需要计算多边形的面法线
    // 顶点是按顺时针方向排列的，u = p0->p1, v=p0->p2, n=uxv
    VECTOR4D u, v, n;

    // 计算 u 和 v
    VECTOR4D_Build(&curr_poly->tvlist[0], &curr_poly->tvlist[1], &u);
    VECTOR4D_Build(&curr_poly->tvlist[0], &curr_poly->tvlist[2], &v);

    //计算叉积
    VECTOR4D_Cross(&u, &v, &n);

    //创建指向视点的向量
    VECTOR4D view;
    VECTOR4D_Build(&curr_poly->tvlist[0], &cam->pos, &view);

    //计算点积
    float dp = VECTOR4D_Dot(&n, &view);

    // 如果 dp< 0，则多边形不可见
    if (dp <= 0.0 )
     SET_BIT(curr_poly->state, POLY4DV1_STATE_BACKFACE);

    } // end for poly

} // end Remove_Backfaces_RENDERLIST4DV1
```

与物体的背面消除一样，对于渲染列表，也应在执行世界坐标到相机坐标变换之前执行背面消除，因为如果多边形是背面（不可见），对其进行上述变换只会浪费 CPU 周期。调用该函数的方式如下：

```
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);
```

调用该函数后，所有属于背面的多边形的 POLY4DV1_STATE_BACKFACE 位都将被设置。

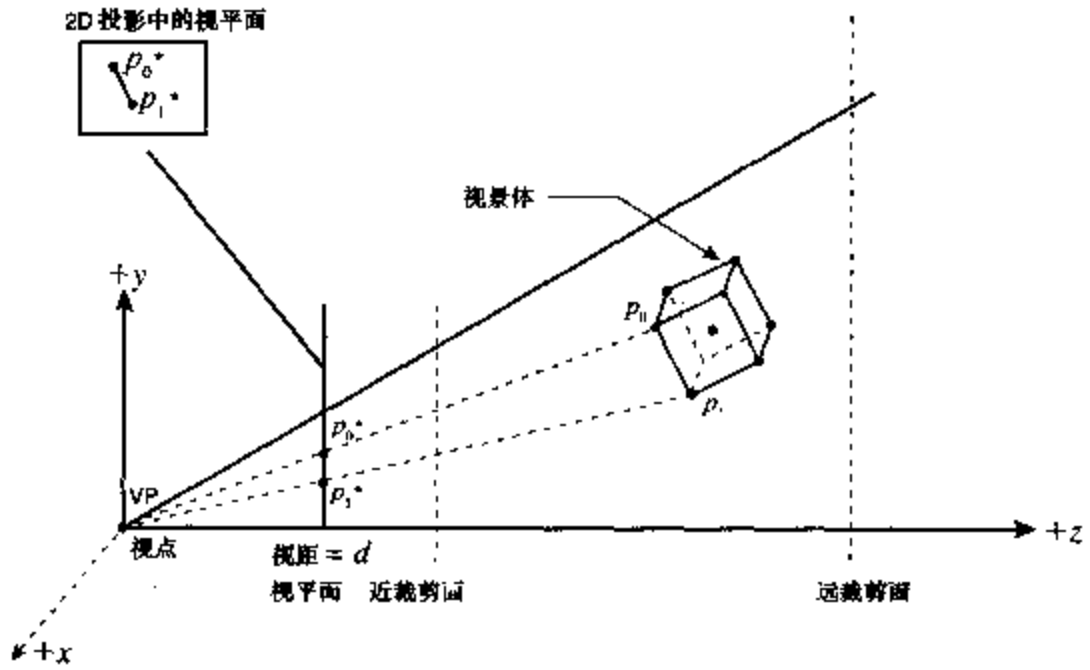
7.3.8　相机坐标到透视坐标变换

3D 流水线中的下一个阶段是相机坐标到透视坐标变换。为方便读者复习，图 7.17 描述了投影数学。简单地说，给定点 $p(x_{world}, y_{world}, z_{world}, 1)$，透视变换为：

```
x_per = viewing_distance*x_world/z_world
y_per = viewing_distance*aspect_ratio*y_world/z_world
```

如果将视距（d）设置为 1.0，则视平面坐标将是归一化的，即坐标 x 和 y 的范围如下：

- x 坐标——−1 到 1；
- y 坐标——$-1/ar$ 到 $1/ar$，其中 ar 为光栅屏幕的宽高比。



$$p_0^* = d * p_{0(x,y)} / p_{0(z)}$$
$$p_1^* = d * p_{1(x,y)} / p_{1(z)}$$

图 7.17　透视投影回顾

另外，如果视平面是方形的，当视距为 1.0 时，x 轴和 y 轴的视野都是 90 度。如果要控制视距或视野（它们之间是相关的），且要投影到任意大小的视平面上，将得到下述投影变换公式：

d = 0.5*viewplane_width*tan(θ_h/2)
x_{per} =(d*x_{world}) / z_{world}
y_{per} =(d*y_{world}*aspect_ratio) / z_{world}

使用代码实现这种变换的方法有两种。手工执行数学运算（速度较快）和使用矩阵执行变换。后一种方法存在的问题是，无法在最后通过矩阵去除以 z，因此要使用 4×4 矩阵变换，必须使用 4D 齐次坐标，并将矩阵变换的结果转换为 3D 坐标。除了需要对齐次坐标进行转换外，矩阵乘法也会浪费 CPU 周期，因此从性能的角度讲，在软件中使用矩阵来执行透视变换是一种糟糕的方法。但是很多硬件实现只使用矩阵。接下来介绍如何使用这两种方法对 OBJECT4DV1 和 RENDERLIST4DV1 进行透视变换。

1. 物体的透视变换

相机对象中存储了执行透视变换所需的所有信息，这包括视距、视平面的宽度和高度、视野（用于计算视距），因此除对每个顶点执行变换运算外，透视变换函数涉及的其他工作并不多。下面是对 OBJECT4DV1 执行透视变换的函数：

```
void Camera_To_Perspective_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据传入的相机对象将物体的相机坐标转换为透视坐标
// 它根本不关心多边形本身
// 而只是对 vlist_trans[]中的顶点进行变换
// 这只是执行透视变换的方法之一，您可能不采用这种方法
// 而是对渲染列表进行变换，因为渲染列表中的多边形表示的是通过了背面剔除的几何体
// 最后，这个函数只是基于实验目的而编写的
// 您可能不会将物体完整地保留到流水线的这个阶段
// 因为物体可能只有一个多边形是可见的，而这个函数对所有多边形都进行变换

// 将物体的每个顶点变换为透视坐标
// 这里假设物体已经被变换为相机坐标，且结果存储在 vlist_trans[]中
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  float z = obj->vlist_trans[vertex].z;

  // 根据相机的观察参数对顶点进行变换
  obj->vlist_trans[vertex].x = cam->view_dist_h *
                obj->vlist_trans[vertex].x/z;
  obj->vlist_trans[vertex].y = cam->view_dist_v *
```

```
                obj->vlist_trans[vertex].y *
                cam->aspect_ratio/z;
  // z 坐标不变

  // 在这个函数中没有使用矩阵，因此无需将分量 x 和 y 除以 w

  } // end for vertex

} // end Camera_To_Perspective_OBJECT4DV1
```

这个函数的代码很少。简单地说，它将每个顶点的 x 坐标和 y 坐标乘以视距，然后除以 z 坐标（从技术的角度说，x 坐标和 y 坐标的处理方式相同，但宽高比使得需要考虑视平面不为方形的情况）。调用该函数的方式如下：

```
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);
```

注意：在这个函数中，没有检查 z 值是否大于 0，这种情况需要在之前的剔除或裁剪步骤中进行处理。这里假设所有顶点都是可投影的。

如果要使用矩阵来执行投影，该如何办呢？可以使用透视矩阵作为参数了调用通用函数 Transform_OBJECT4DV1()，而透视变换矩阵可以使用下面的函数来创建：

```
void Build_Camera_To_Perspective_MATRIX4X4(CAM4DV1_PTR cam,
                      MATRIX4X4_PTR m)
{
// 这个函数创建相机坐标到透视坐标变换矩阵
// 在大多数情况下，相机的视平面是归一化的（2×2），FOV 为 90 度
// 这里假设使用的是 4D 齐次坐标，在某个时候将执行 4D 坐标到 3D 坐标转换
// 这种操作可能在透视变换之后马上进行，也可能在屏幕变换之后才进行

Mat_Init_4X4(m,
cam->view_dist_h, 0,                    0, 0,
0,        cam->view_dist_v*cam->aspect_ratio, 0, 0,
0,        0,                    1, 1,
0,        0,                    0,0);

} // end Build_Camera_To_Perspective_MATRIX4X4
```

创建透视变换矩阵后，只需这样将变换应用于 OBJECT4DV1 即可：

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);
Transform_OBJECT4DV1(&obj, &mper, TRANSFORM_TRANS_ONLY,1);
```

当然，还有一个细节需要处理。执行透视变换后，顶点的 w 坐标不再为 1.0；此时的坐标为齐次坐标，使用之前必须将其转换为非齐次坐标。为此，只需除以 w 分量即可。下面是对 OBJECT4DV1 执行这项任务的函数：

```
void Convert_From_Homogeneous4D_OBJECT4DV1(OBJECT4DV1_PTR obj)
{
// 这个函数将变换后的顶点列表中所有的顶点从 4D 齐次坐标转换为 3D 坐标
// 方法是将分量 x、y 和 z 都除以 w

for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  // 转换为非齐次坐标
  VECTOR4D_DIV_BY_W(&obj->vlist_trans[vertex]);
  } // end for vertex

} // end Convert_From_Homogeneous4D_OBJECT4DV1
```

因此，要得到真正的 3D 坐标，只需这样调用上述函数：

```
Convert_From_Homogeneous4D_OBJECT4DV1(&obj);
```

2. 渲染列表的透视变换

REDNERLIST4DV1 的透视变换方式与 OBJECT4DV1 相同，但访问数据结构的方式稍有不同。下面是

不使用矩阵来执行相机坐标到透视坐标变换的函数：

```
void Camera_To_Perspective_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                          CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据传入的相机对象，将渲染列表中的每个多边形都变换为透视坐标
// 如果在流水线的上游已经将物体转换为多边形，并将这些多边形插入到渲染列表中
// 将使用这个函数而不是基于物体的函数来执行透视变换

// 将渲染列表中的每个多边形变换为透视坐标
// 这里假设渲染列表中的多变形已被变换为相机坐标
// 且变换结果存储在多边形的 tvlist[]中

for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// 获取当前多边形
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// 该多边形是否有效?
// 当且仅当多边形没有被裁剪和剔除掉、处于活动状态并可见时，才对其进行变换
// 但在线框引擎中，"背面"的概念无关紧要
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形
// 满足条件，对其进行变换
for (int vertex = 0; vertex < 3; vertex++)
  {
  float z = curr_poly->tvlist[vertex].z;

  // 根据相机的观察参数对顶点进行变换
  curr_poly->tvlist[vertex].x = cam->view_dist_h *
                curr_poly->tvlist[vertex].x/z;
  curr_poly->tvlist[vertex].y = cam->view_dist_v *
                curr_poly->tvlist[vertex].y *
                cam->aspect_ratio/z;
  // z 坐标不变

  // 在这个函数中没有使用矩阵，因此无需将分量 x 和 y 除以 w

  } // end for vertex

} // end for poly

} // end Camera_To_Perspective_RENDERLIST4DV1
```

该函数的调用方式与相应的 OBJECT4DV1 版函数相同：

```
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);
```

要使用矩阵来执行渲染列表的透视变换，可调用函数 Build_Camera_To_Perspective_MATRIX4X4()创建透视变换矩阵，然后调用函数 Transform_RENDERLIST4DV1()，如下所示：

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);

Transform_RENDERLIST4DV1(&rend_list, &mper, TRANSFORM_TRANS_ONLY);
```

同样，这里也需要将齐次坐标转换为非齐次坐标，因为 w 不再为 1.0（实际上，$w=z$）。将 4D 齐次坐标转换为 3D 坐标的函数如下：

```
void Convert_From_Homogeneous4D_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list)
{
// 这个函数将变换后的顶点列表中所有有效的多边形顶点从 4D 齐次坐标转换为 3D 坐标
// 方法是将分量 x、y 和 z 都除以 w
for (int poly = 0; poly < rend_list->num_polys; poly++)
```

```
{
// 获取当前多边形
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// 该多边形是否有效?
// 当且仅当多边形没有被裁剪和剔除掉、处于活动状态并可见时，才对其进行变换
// 但在线框引擎中，"背面"的概念无关紧要
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形

// 满足条件，进行变换
for (int vertex = 0; vertex < 3; vertex++)
  {
  // 转换为非齐次坐标
  VECTOR4D_DIV_BY_W(&curr_poly->tvlist[vertex]);
  } // end for vertex

} // end for poly

} // end Convert_From_Homogeneous4D_RENDERLIST4DV1
```

因此，对渲染列表执行透视变换时，需要按如下顺序调用各个函数：

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);

Transform_RENDERLIST4DV1(&rend_list, &mper, TRANSFORM_TRANS_ONLY);

Convert_From_Homogeneous4D_RENDERLIST4DV1(&rend_list);
```

现在，我们得到了一个位于视平面上的虚拟图像。接下来需要将其投影到屏幕（视口）上，这是 3D 流水线中除光栅化外的最后一个步骤。

7.3.9　透视坐标到屏幕（视口）坐标变换

从某种意义上说，透视坐标到屏幕坐标变换是 3D 流水线的最后一个阶段，它将视平面坐标进行缩放，变成屏幕坐标，如图 7.18 所示。在这种变换中必须考虑这样一点，即大多数光栅屏幕的原点位于左上角，y 轴的方向与标准 2D 迪卡尔坐标系相反。当然，如果在透视变换中，视平面的大小与视口相同，则无需执行缩放操作，但在大多数情况下，需要执行平移，并反转 y 轴，因为在投影时，我们假设视平面的中心为原点，其$+x$ 轴指向右方，$+y$ 轴指向上方，而光栅屏幕的原点位于左上角，y 轴方向与此相反。因此，无论在什么情况下，都需要执行某种形式的视口变换。

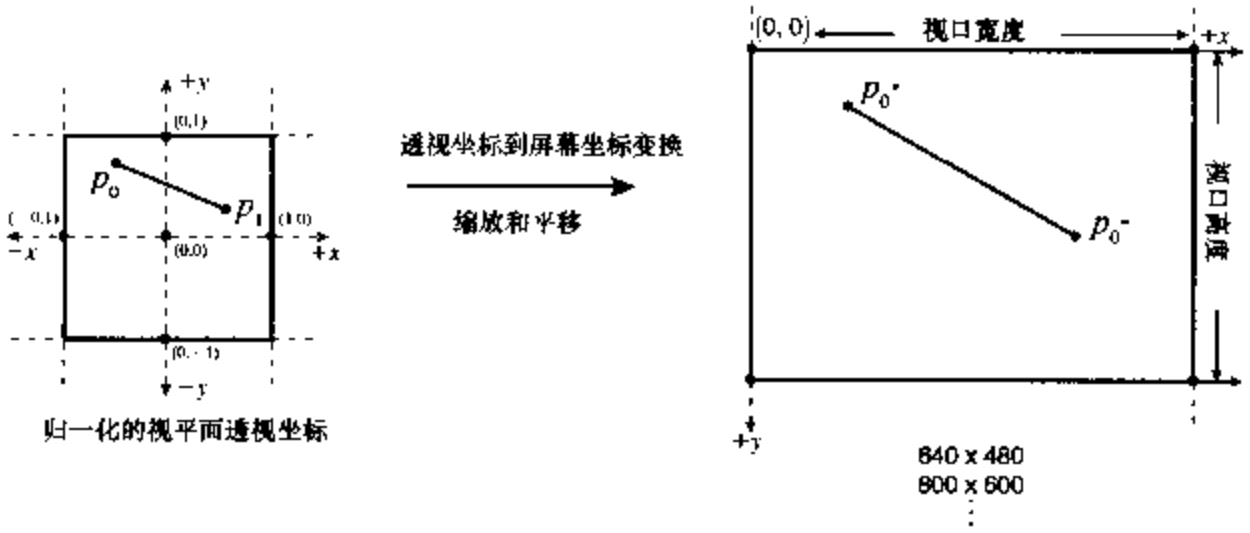


图 7.18　透视坐标到屏幕坐标变换

接下来介绍用于物体和渲染列表的视口变换函数。需要指出的是，这些函数假设透视变换是归一化的，即对于方形投影，投影到一个 2×2 的视平面上，对于非方形投影，投影到一个 $2\times 2/ar$ 的视平面上；在这种情况下，必须执行缩放操作。下一节将讨论如何将透视变换和屏幕变换合而为一。

1. 物体的视口变换

首先来看一个手工执行透视坐标到屏幕（视口）坐标变换的函数，这个函数假设透视坐标是归一化的，因此必须根据屏幕（视口）的大小对透视坐标进行缩放和平移，并反转 y 轴。该函数如下：

```
void Perspective_To_Screen_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据传入的视口信息将物体的透视坐标变换为屏幕坐标
// 但完全不关心多边形本身
// 而只是对 vlist_trans[]中的顶点进行变换
// 这只是执行屏幕变换的方法之一，您可能不采用这种方法
// 而是对渲染列表进行变换，因为渲染列表中的多边形表示的是通过了背面剔除的几何体
// 最后，这个函数只是基于实验目的而编写的
// 您可能不会将物体完整地保留到流水线的这个阶段
// 因为物体可能只有一个多边形是可见的，而这个函数对所有多边形都进行变换
// 对物体执行透视变换后调用这个函数

// 将物体的每个顶点变换为屏幕坐标
// 这里假设物体已经被变换为透视坐标，且结果存储在 vlist_trans[]中

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  // 假设物体的透视坐标为归一化的，取值范围为-1 到 1
  // 根据视口大小对坐标进行缩放，并反转 y 轴
  obj->vlist_trans[vertex].x = alpha + alpha*obj->vlist_trans[vertex].x;
  obj->vlist_trans[vertex].y = beta - beta *obj->vlist_trans[vertex].y;

  } // end for vertex

} // end Perspective_To_Screen_OBJECT4DV1
```

这个函数将归一化的视平面坐标（x 和 y 的取值范围都是-1 到 1）映射到光栅屏幕上。需要指出的是，无需对 4D 齐次坐标进行转换，因为在相机坐标到透视坐标变换期间，已经将 x 和 y 坐标除以了 z。因此经过上述变换后，每个顶点的分量 x 和 y 表示的已经是光栅屏幕上的位置。

有可能在透视变换期间没有除以 z，即这种变换是使用矩阵完成的，且没有对得到的 4D 齐次坐标（w 不等于 1.0）进行转换。在这种情况下，仍可以使用上述函数，但在某个地方将每个顶点的坐标除以 w 分量，将齐次坐标转换为非齐次坐标。然而，在大多数情况下，都会在视口变换之前对 4D 齐次坐标进行转换，虽然没有规定必须这样做。要对物体的透视坐标进行变换，可以这样调用上述函数：

```
// 首先执行相机坐标到透视坐标变换
// 将坐标转换为非齐次坐标后
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);

// 执行视口变换
Perspective_To_Screen_OBJECT4DV1(&obj, &cam);
```

接下来讨论如何使用矩阵来执行透视坐标到屏幕坐标变换。使用的矩阵随透视坐标是 4D 齐次坐标还是已经通过除以 w 或 z（取决于透视变换方式）转换为 3D 坐标（从技术上说是 2D 坐标，因为我们只需要 x 和 y 坐标）而异。这里将对这两种情况都进行讨论。

假设使用下述代码执行了透视变换：

```
MATRIX4X4 mper;
Build_Camera_To_Perspective_MATRIX4X4(&cam, &mper);

Transform_OBJECT4DV1(&obj, &mper, TRANSFORM_TRANS_ONLY,1);
```

结果将为 4D 齐次坐标。要计算每个顶点的 3D 坐标，需要将其除以 w 的值。然而，可以将这项工作推迟到后面去完成，现在仍对 4D 坐标执行透视坐标到屏幕（视口）坐标变换。在这种情况下，可以使用

下述函数来创建变换矩阵：

```
void Build_Perspective_To_Screen_4D_MATRIX4X4(CAM4DV1_PTR cam, MATRIX4X4_PTR m)
{
// 这个函数创建透视坐标到屏幕坐标变换矩阵
// 这里假定要对齐次坐标进行变换，并在光栅化期间执行 4D 齐次坐标到 3D 坐标转换
// 当然执行 2D 渲染时只考虑点的 x 坐标和 y 坐标
// 这个函数创建的矩阵用于对透视坐标进行缩放和平移，将其变换为屏幕坐标
// 创建矩阵时假定透视坐标是归一化的，即视平面大小为 2×2
// x 坐标的取值范围为-1 到 1，y 坐标的取值范围为-1/ar 到 1/ar

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);
Mat_Init_4X4(m, alpha,  0,   0,  0,
        0,   -beta, 0,  0,
        alpha,  beta, 1,  0,
        0,    0,    0,  1);

} // end Build_Perspective_To_Screen_4D_MATRIX4X4()
```

因此，可以这样创建变换矩阵并对物体进行变换：

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_4D_MATRIX4X4 (&cam, &mscr);
Transform_OBJECT4DV1(&obj, &mscr, TRANSFORM_TRANS_ONLY,1);
```

现在，只需将齐次坐标变换为 3D 坐标，便可以得到屏幕坐标：

```
Convert_From_Homogeneous4D_OBJECT4DV1(&obj);
```

提示：这些内容不那么好懂，读者必须牢记的是，无法用矩阵来完成除以 z 的工作，因此必须将 z 转移到 w 分量中，然后通过除以 w 来实现透视变换（非几何变换）。也就是说，问题将涉及 4 个分量，这很好，因为这样就可以使用 4×4 矩阵。然而，在大多数情况下，作者喜欢将 w 设置为 1，并手工执行透视变换和除以 z 的任务，而不是增加一个将齐次坐标转换为 3D 坐标的步骤（将其他所有坐标分量都除以 w）。

如果物体的所有顶点都已经从 4D 齐次坐标转换为 3D 坐标，则只需编写一个功能与 Perspective_To_Screen_OBJECT4DV1()相同的使用矩阵执行视口变换的函数，即这个函数假设坐标已经是 3D 的，w=1。下面的函数用于创建对 3D 坐标进行透视变换的矩阵：

```
void Build_Perspective_To_Screen_MATRIX4X4(CAM4DV1_PTR cam, MATRIX4X4_PTR m)
{
// 这个函数创建透视坐标到屏幕坐标变换矩阵
// 这里假设透视坐标已经从 4D 齐次坐标转换为 3D 坐标
// 这个函数创建的矩阵用于对透视坐标进行缩放和平移，将其变换为屏幕坐标
// 创建矩阵时假定透视坐标是归一化的，即视平面大小为 2×2
// x 坐标的取值范围为-1 到 1，y 坐标的取值范围为-1/ar 到 1/ar
// 这个函数与前一个函数的惟一差别在于，
// 矩阵的最后一列没有将 w 设置为 z
// 事实上，w 的值已无关紧要，因为应用这个矩阵之前，所有坐标都已经从 4D 转换为 3D

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

Mat_Init_4X4(m, alpha,    0,   0,  0,
        0,   -beta, 0,  0,
        alpha,  beta, 1,  0,
        0,    0,   0,   1);

} // end Build_Perspective_To_Screen_MATRIX4X4()
```

这个函数假设顶点坐标已经是 3D 的（从技术上说是 2D 的，只是归一化的）。要对物体进行视口变换，可以这样做：

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_MATRIX4X4(&cam, &mscr);
Transform_OBJECT4DV1(&obj, &mscr, TRANSFORM_TRANS_ONLY,1);
```

这些范例旨在表明，将坐标从 4D 转换为 3D 后（通常是在透视变换期间完成的），流水线中的数学运算和矩阵也将发生变换，因此需要提供两个分支（branch）。就个人而言，作者喜欢在透视变换期间将透视坐标从 4D 转换为 3D。

2. 渲染列表的视口变换

渲染列表的视口变换与物体相同，只是访问数据结构的方式不同而已。首先来看透视变换为归一化的（已经使用 z 除以坐标 x 和 y）情况，在这种情况下，只需将透视坐标进行缩放和平移，便可得到屏幕坐标。函数如下：

```
void Perspective_To_Screen_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                          CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据传入的相机的视口参数将渲染列表中的透视坐标变换为屏幕坐标
// 这里假设透视坐标是归一化的
// 如果在流水线的上游已经将物体转换为多边形，并将这些多边形插入到渲染列表中
// 则应调用这个函数，而不是基于物体的屏幕变换函数
// 仅当之前执行的是归一化透视变换时才使用这个函数
// 将渲染列表中的每个多边形从透视坐标变换为屏幕坐标
// 这里假设已经对渲染列表执行归一化透视变换，且结果存储在 tvlist[]中
for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// 获得当前多边形
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// 该多边形是否有效？
// 当且仅当多边形没有被裁剪和剔除掉、处于活动状态并可见时，才对其进行变换
// 但在线框引擎中，"背面"的概念无关紧要
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
   (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
   (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

// 满足条件，对其进行变换
for (int vertex = 0; vertex < 3; vertex++)
  {
  // 顶点的透视坐标是归一化的，取值范围为-1 到 1
  // 对坐标进行缩放，并反转 y 轴

  // 根据相机的视口参数对顶点进行变换
  curr_poly->tvlist[vertex].x = alpha + alpha*curr_poly->tvlist[vertex].x;
  curr_poly->tvlist[vertex].y = beta - beta *curr_poly->tvlist[vertex].y;
  } // end for vertex

} // end for poly

} // end Perspective_To_Screen_RENDERLIST4DV1
```

这个函数只是对坐标进行缩放，并反转 y 轴。下面是一个使用该函数的例子。在这个例子中，首先调用了一个结果为非齐次坐标的透视变换函数。

```
// 执行透视变换，并将 x、y 坐标除以 z
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// 将透视坐标变换为屏幕坐标
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);
```

现在，读者应该知道了 4D 坐标和 3D 坐标之间的差别。和物体一样，也可以使用矩阵来执行渲染列表的视口变换，在这种情况下，需要考虑透视坐标是 3D 的还是 4D 的。透视坐标为 4D 时应这样做：

```
MATRIX4X4 mscr;
Build_Perspective_To_Screen_4D(&cam, &mscr);
```

```
Transform_RENDERLIST4DV1(&rend_list, &mscr, TRANSFORM_TRANS_ONLY);
```

现在，只需调用下述函数，将 4D 齐次坐标转换为 3D 坐标，便可得到屏幕坐标：

```
Convert_From_Homogeneous4D_RENDERLIST4DV1(&rend_list);
```

如果透视坐标是 3D 的，则不用执行 4D 齐次坐标到 3D 坐标转换，可以像前面介绍过的那样执行简单的透视坐标到屏幕坐标变换：

```
MATRIX4X4 mscr;

Build_Perspective_To_Screen_MATRIX4X4(&cam, &mscr);

Transform_RENDERLIST4DV1(&rend_list, &mscr, TRANSFORM_TRANS_ONLY);
```

正如读者看到的，矩阵的优点在于，它是通用的。使用矩阵对物体和渲染列表进行视口变换时，惟一的差别在于，最后调用的变换函数 Transform_*() 不同。

提示：在实际编程时，可以将 Transform_*() 声明为虚函数（而物体和渲染列表都是类），这样便可以调用单个函数，该函数将根据底层对象的类型执行相应的变换。这些工作将在本书后面完成。

7.3.10　合并透视变换和屏幕变换

接下来基于一些常识进行优化。作者将介绍完成工作的所有方式，让读者知道所以然。现在，读者应该已经认识到，矩阵虽然很好，但绝对不适合用于透视变换，手工执行透视变换更容易。另外，在很多情况下，使用不同的函数分别执行透视变换和屏幕变换是合理的，但如果要将图像存储到缓存中，则没有必要这样做，因为屏幕变换函数只是对透视坐标进行缩放和平移而已。因此，通常将这两种操作合而为一，创建一个同时执行透视变换和屏幕（视口）变换的函数。

1. 物体的相机坐标到屏幕坐标变换

一般而言，执行相机坐标到屏幕坐标变换时，最快捷的方式是创建一个手工完成这项工作并将 x 和 y 坐标除以 z 的函数。下面是用于物体的函数：

```
void Camera_To_Perspective_Screen_OBJECT4DV1(OBJECT4DV1_PTR obj, CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据传入的相机对象的 view_dist_h 和 view_dist_v（应根据视口宽度和高度相应地设置）
// 将物体的相机坐标变换为屏幕坐标
// 它完全不考虑多边形本身
// 而只是对 vlist_trans[]中的顶点进行变换
// 这只是执行相机坐标到屏幕坐标变换的方法之一，您可能不采用这种方法
// 而是对渲染列表进行变换，因为渲染列表中的多边形表示的是通过了背面剔除的几何体
// 这个函数只是基于实验目的而编写的
// 您可能不会将物体完整地保留到流水线的这个阶段
// 因为物体可能只有一个多边形是可见的，而这个函数对所有多边形都进行变换
// 最后，这个函数对 y 轴进行反转，以确保生成的坐标是屏幕坐标，可用于渲染

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

// 将物体的每个顶点变换为屏幕坐标
// 这里假设物体已被变换为相机坐标，且结果存储在 vlist_trans[]中
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  float z = obj->vlist_trans[vertex].z;

  // 根据相机的观察参数对顶点进行变换
  obj->vlist_trans[vertex].x = cam->view_dist_h *
                obj->vlist_trans[vertex].x/z;
  obj->vlist_trans[vertex].y = cam->view_dist_v *
                obj->vlist_trans[vertex].y *
                cam->aspect_ratio/z;
  // z 坐标不变
```

```
    // 在这个函数中没有使用矩阵，因此无需将分量 x 和 y 除以 w

    // 现在坐标的范围如下
    // x:(-viewport_width/2 to viewport_width/2)
    // y:(-viewport_height/2 to viewport_height/2),
    // 接下来需要对坐标进行平移，并反转 y 轴
    // 以得到屏幕坐标
    obj->vlist_trans[vertex].x = obj->vlist_trans[vertex].x + alpha;
    obj->vlist_trans[vertex].y = -obj->vlist_trans[vertex].y + beta;

    } // end for vertex

} // end Camera_To_Perspective_Screen_OBJECT4DV1
```

这个函数减少了更多的数学计算。我们无需根据视口的大小相应地缩放视平面坐标，因为物体被投影到大小为 viewport_width×viewport_height 的视平面上，这进一步减少了计算量。调用上述函数后，vlist_trans[ ]中的坐标便可以用来绘图。下面是一个调用该函数的例子：

```
Camera_To_Perspective_Screen_OBJECT4DV1(&obj,&cam);
```

作者也不知道自己为何一直列举调用这些极其简单的函数的范例，但这样做没有害处。需要指出的是，调用上述函数之前，坐标必须是相机坐标，仅此而已。

2. 渲染列表的相机坐标到屏幕坐标变换

对渲染列表一次性执行相机坐标到屏幕坐标变换也没有什么神奇之处，只是访问数据结构的方式稍有不同而已。下面是执行这种变换的函数：

```
void Camera_To_Perspective_Screen_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list,
                          CAM4DV1_PTR cam)
{
// 这个函数不是基于矩阵的
// 它根据传入的相机对象的 view_dist_h 和 view_dist_v（应根据视口宽度和高度相应地设置）
// 将渲染列表中的相机坐标变换为屏幕坐标
// 它只是对 tvlist[]中的顶点进行变换
// 最后，这个函数对 y 轴进行反转，以确保生成的坐标是屏幕坐标，可用于渲染

// 将渲染列表的每个多边形变换为屏幕坐标
// 这里假设渲染列表已被变换为相机坐标，且结果存储在 tvlist []中
for (int poly = 0; poly < rend_list->num_polys; poly++)
{
// 获取当前多边形
POLYF4DV1_PTR curr_poly = rend_list->poly_ptrs[poly];

// 该多边形是否有效？
// 当且仅当多边形没有被裁剪和剔除掉、处于活动状态并可见时，才对其进行变换
// 但在线框引擎中，"背面"的概念无关紧要
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
    (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
    (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形

float alpha = (0.5*cam->viewport_width-0.5);
float beta = (0.5*cam->viewport_height-0.5);

// 满足条件，对其进行变换
for (int vertex = 0; vertex < 3; vertex++)
  {
  float z = curr_poly->tvlist[vertex].z;

  // 根据相机的观察参数对顶点进行变换
  curr_poly->tvlist[vertex].x = cam->view_dist_h *
                curr_poly->tvlist[vertex].x/z;
  curr_poly->tvlist[vertex].y = cam->view_dist_v *
                curr_poly->tvlist[vertex].y *
```

```
                    cam->aspect_ratio/z;
    // z 坐标不变

    // 在这个函数中没有使用矩阵，因此无需将分量 x 和 y 除以 w

    // 现在坐标的范围如下
    // x:(-viewport_width/2 to viewport_width/2)
    // y:(-viewport_height/2 to viewport_height/2),
    // 接下来需要对坐标进行平移，并反转 y 轴
    // 以完成屏幕变换
    curr_poly->tvlist[vertex].x = curr_poly->tvlist[vertex].x + alpha;
    curr_poly->tvlist[vertex].y = -curr_poly->tvlist[vertex].y + beta;

    } // end for vertex

} // end for poly

} // end Camera_To_Perspective_Screen_RENDERLIST4DV1
```

下面是一个调用该函数的例子：

```
Camera_To_Perspective_Screen_RENDERLIST4DV1(&rend_list,
&cam);
```

至此，3D 流水线的大部分工作已经完成。也许惟一让读者感到迷惑的地方是，进行透视变换时，必须决定采用手工方式还是矩阵方式以及是否继续使用 4D 坐标。另一方面，可以一次性执行透视变换和屏幕变换，以节省时间和计算量。在大多数情况下，我们将采用这种合而为一的方法，至少使用手工方式来执行透视变换和屏幕变换，将坐标转换为 3D 的，因为不想将 4D 坐标到 3D 坐标转换的工作留到流水线的后续阶段去完成。然而，正如前面指出的，使用硬件来实现引擎时，我们可能别无选择，只能使用矩阵和 4D 坐标。

7.4　渲染 3D 世界

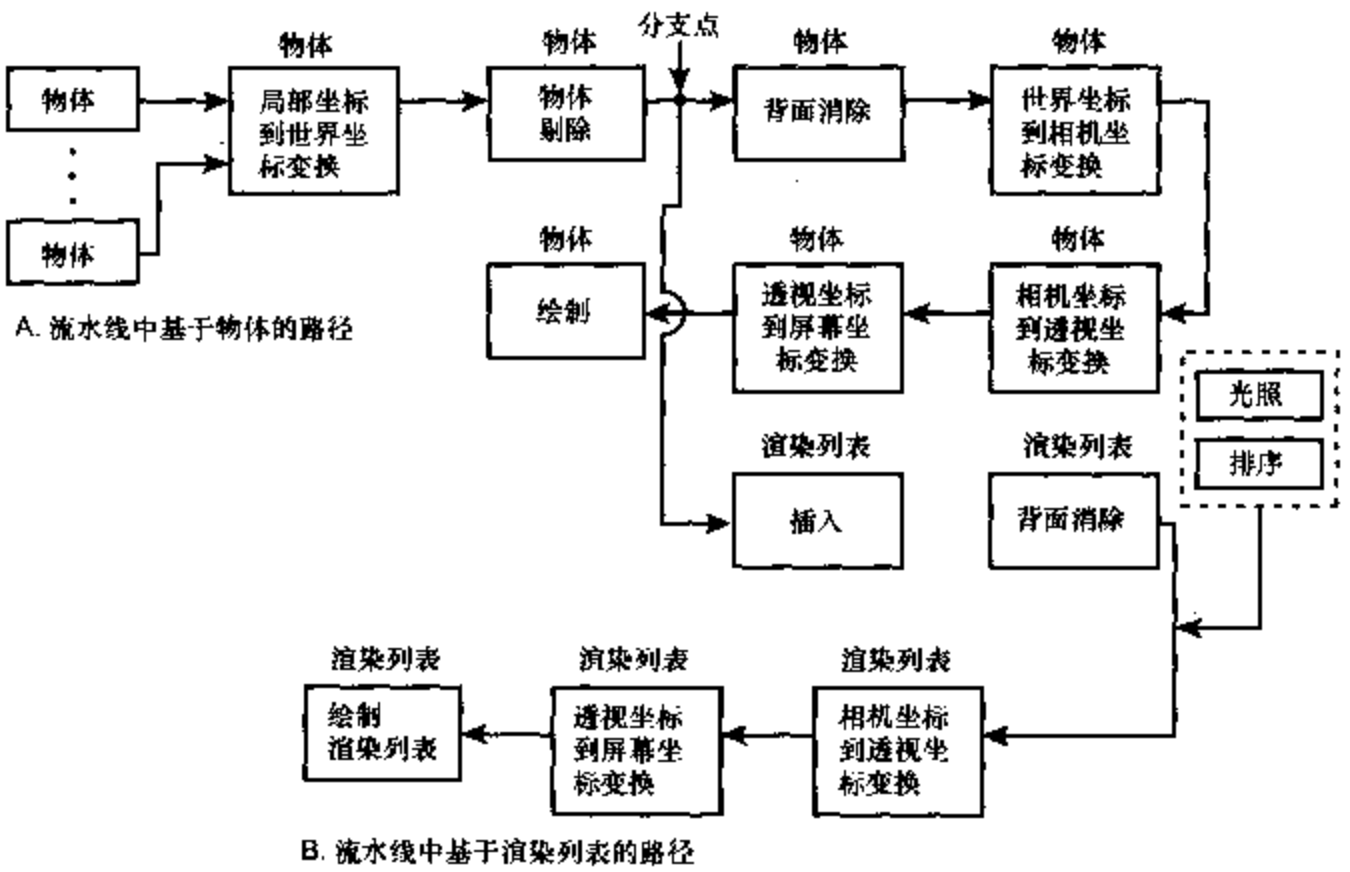
终于可以渲染 3D 世界了！作者面临的困境在于，在本章前面巨大的篇幅中，不能介绍任何演示程序，因为首先必须让读者理解流水线的每个步骤，并编写相应的代码。现在，这些准备工作已经完成，可以编写演示程序。不过，先回顾一下 3D 流水线，然后再编写一些简单的图形函数，根据 OBJECT4DV1 或 RENDERLIST4DV1 来绘制线框网格。

现在的 3D 流水线

我们已经编写了从磁盘中加载 3D 物体、将其存储到 OBJECT4DV1 结构中以及在流水线中处理一个或多个物体所需的代码。如果愿意，也可以在 3D 流水线中这样处理物体：将其分解为多边形，然后将它们插入到一个主多边形列表（数据结构 RENDERLIST4DV1）中。采用哪种方法完全取决于您，这两种方法各有利弊。一般而言，大多数 3D 引擎都同时使用了这两种方法。对于运动的东西，将其作为物体进行处理；但对于属于大型网格的环境，则将其分解，并将各个区段插入到渲染列表中。为简化起见，我们现在假设游戏世界是由物体构成的，没有需要处理的内部环境和外部环境。

图 7.19 说明几何体可能经历的各种路径，但通常必须经过下述阶段：

阶段 0　加载并放置物体；
阶段 1　局部坐标到世界坐标变换；
阶段 2　物体剔除和背面消除；
阶段 3　世界坐标到相机坐标变换；
阶段 4　相机坐标到透视坐标变换；
阶段 5　透视坐标到屏幕坐标变换；
阶段 6　渲染几何体。



A. 流水线中基于物体的路径

B. 流水线中基于渲染列表的路径

图 7.19　观察流水线和多条渲染路径

可能在阶段 2 之后的某个地方将物体转换为多边形，因为您希望至少在物体未被转换前对其进行剔除。将物体转换为多边形后，就失去了它们的结构，不能再将它们视为物体。我们没有进行 3D 裁剪，但只要执行了物体剔除便是安全的，因为在物体剔除操作中，至少根据近裁剪面对物体进行了裁剪。在很大程度上说，不会有顶点的 z 坐标小于 0 的几何体，但需要进行 3D 裁剪，这将在下一章介绍。就现在而言，假设所有被投影的顶点都是安全的，如果它们确实位于 2D 视口之外，2D 绘图函数将把它们裁剪掉。

不管在 3D 流水线中采取哪条路径，都将在某个时刻有下列两样东西之一：一个要渲染的 OBJECT4DV1 列表和一个表示所有要渲染的多边形的 RENDERLIST4DV1。当然，可以使用多个渲染列表，但就现在而言，我们假设只有一个渲染列表，它包含所有物体的多边形。因此，需要编写两个函数，一个能够绘制 OBJECT4DV1 中的所有多边形，另一个能够绘制 RENDERLIST4DV1 中的所有多边形。

绘制 OBJECT4DV1——编写函数之前，需要指出的一点是，我们将使用线框图形，不执行隐藏线消除，也不进行排序，这意味着物体是可以看穿的。另外物体的绘制顺序可能是错误的，因为我们没有根据物体的深度对其进行排序，也没有使用其他排序方法。物体是空心的。因此，我们只需编写一个这样的函数：假设在变换后的物体顶点列表中，包含的是有效的屏幕坐标，并依次绘制每个三角形，这个函数如下所示。这个函数检查每个多边形是否可见、被裁剪掉或是背面。

```
void Draw_OBJECT4DV1_Wire16(OBJECT4DV1_PTR obj,
                UCHAR *video_buffer, int lpitch)
{
// 这个函数使用线框模式和 16 位 RGB 颜色模式将物体渲染到屏幕上
// 它根本没有考虑隐藏线消除等问题
// 这是一种在不将物体转换为多边形的情况下对其进行渲染的简易方式
// 这里假设坐标为屏幕坐标，并执行 2D 裁剪

// 遍历物体的多边形列表，并渲染每个多边形
for (int poly=0; poly < obj->num_polys; poly++)
  {
  // 当且仅当多边形处于活动状态并可见时才渲染它
  // 在线框引擎中，"背面"的概念无关紧要
  if (!(obj->plist[poly].state & POLY4DV1_STATE_ACTIVE) ||
     (obj->plist[poly].state & POLY4DV1_STATE_CLIPPED ) ||
     (obj->plist[poly].state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入到下一个多边形

  // 提取指向顶点列表的顶点索引
  // 多边形不是自包含的，而是基于物体的顶点列表的
```

```
    int vindex_0 = obj->plist[poly].vert[0];
    int vindex_1 = obj->plist[poly].vert[1];
    int vindex_2 = obj->plist[poly].vert[2];

    // 绘制线段
    Draw_Clip_Line16(obj->vlist_trans[ vindex_0 ].x,
              obj->vlist_trans[ vindex_0 ].y,
              obj->vlist_trans[ vindex_1 ].x,
              obj->vlist_trans[ vindex_1 ].y,
              obj->plist[poly].color,
              video_buffer, lpitch);

    Draw_Clip_Line16(obj->vlist_trans[ vindex_1 ].x,
              obj->vlist_trans[ vindex_1 ].y,
              obj->vlist_trans[ vindex_2 ].x,
              obj->vlist_trans[ vindex_2 ].y,
              obj->plist[poly].color,
              video_buffer, lpitch);

    Draw_Clip_Line16(obj->vlist_trans[ vindex_2 ].x,
             obj->vlist_trans[ vindex_2 ].y,
             obj->vlist_trans[ vindex_0 ].x,
             obj->vlist_trans[ vindex_0 ].y,
             obj->plist[poly].color,
             video_buffer, lpitch);

    } // end for poly

} // end Draw_OBJECT4DV1_Wire
```

使用虚拟计算机系统和图形缓存，可以只顾埋头编写 3D 函数，而无需了解硬件——惟一需要知道的是视频缓存和内存块的地址。该函数遍历多边形列表，并绘制组成 3D 物体的所有三角形，它使用一个 2D 函数来绘制裁剪后的直线。设置好图形系统，创建一个物体对象 obj 并执行完 3D 流水线中的操作后，便可以这样调用这个函数：

```
Draw_OBJECT4DV1_Wire16(&obj, video_buffer,lpitch);
```

绘制 RENDERLIST4DV1——绘制渲染列表的函数与绘制物体的函数相同，只是访问数据结构的方式不同而已。这个函数如下：

```
void Draw_RENDERLIST4DV1_Wire16(RENDERLIST4DV1_PTR rend_list,
                UCHAR *video_buffer, int lpitch)
{
// 这个函数"执行"渲染列表
// 即使用线框模式和 16 位 RGB 颜色模式绘制渲染列表中所有的面
// 在线框模式下，无需对多边形进行排序，但后面需要这样做，以消除隐藏面
// 另外，这里让函数去判断位深，并调用相应的光栅化器

// 现在，我们只有一个多边形列表，可以绘制它们
// to draw them
for (int poly=0; poly < rend_list->num_polys; poly++)
  {
  // 当且仅当多边形处于活动状态且可见时才渲染它
  // 但在线框引擎中，"背面"概念无关紧要
  if (!(rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_ACTIVE) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_CLIPPED ) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形

  // 绘制三角形的边
  //2D 初始化过程中已经设置好裁剪，位于 2D 屏幕/窗口外的多边形都将被裁剪掉
  Draw_Clip_Line16(rend_list->poly_ptrs[poly]->tvlist[0].x,
        rend_list->poly_ptrs[poly]->tvlist[0].y,
        rend_list->poly_ptrs[poly]->tvlist[1].x,
        rend_list->poly_ptrs[poly]->tvlist[1].y,
        rend_list->poly_ptrs[poly]->color,
```

```
            video_buffer, lpitch);

      Draw_Clip_Line16(rend_list->poly_ptrs[poly]->tvlist[1].x,
            rend_list->poly_ptrs[poly]->tvlist[1].y,
            rend_list->poly_ptrs[poly]->tvlist[2].x,
            rend_list->poly_ptrs[poly]->tvlist[2].y,
            rend_list->poly_ptrs[poly]->color,
            video_buffer, lpitch);

      Draw_Clip_Line16(rend_list->poly_ptrs[poly]->tvlist[2].x,
            rend_list->poly_ptrs[poly]->tvlist[2].y,
            rend_list->poly_ptrs[poly]->tvlist[0].x,
            rend_list->poly_ptrs[poly]->tvlist[0].y,
            rend_list->poly_ptrs[poly]->color,
            video_buffer, lpitch);

      } // end for poly

   } // end Draw_RENDERLIST4DV1_Wire
```

这个函数的调用方式也没什么不同，只是需要一个渲染列表而不是一个物体：

```
Draw_RENDERLIST4DV1_Wire16(&rend_list, video_buffer,lpitch);
```

至此，为编写演示程序做好了准备。编写好所有的流水线和渲染算法后，接下来将编写一些演示程序，以复习本章前面介绍的各种技术。下一节将通过一些范例让读者深入了解如何完成各项工作。

7.5 3D 演示程序

本节介绍一些完整的演示程序。由于篇幅有限，这里只介绍一些使用前面编写的函数创建的基本线框范例。正如作者指出的，阅读本章后，读者将能够轻松地编写出像 Battle Zone 这样的游戏。对于每个演示程序，都列出了其名称及说明，以便读者可以快速浏览本节，从中找到自己感兴趣的内容。另外，这里没有列出演示程序的完整代码，而只列出了主循环中的重要代码，因为它们使用的函数都已经在前面介绍过。要编译这些演示程序，除演示程序主.CPP 文件外，还需要下述文件：

- T3DLIB1.CPP|H；
- T3DLIB2.CPP|H；
- T3DLIB3.CPP|H；
- T3DLIB4.CPP|H；
- T3DLIB5.CPP|H；
- DDRAW.LIB；
- DSOUND.LIB；
- DINPUT.LIB；
- DINPUT8.LIB。

所有.PLG 文件都存储在附带光盘上本章对应的目录中。

7.5.1 单个 3D 三角形

- 加载的物体：没有，手工创建一个多边形；
- 相机类型：欧拉相机，位置固定，视野为 90 度，视平面是归一化的；
- 投影类型：先进行透视变换，然后进行视口变换；
- 3D 流水线使用的计算方法：所有函数都使用手工计算，没有变换是基于矩阵的；
- 背面消除：没有执行；
- 物体剔除：没有执行；
- 渲染方式：包含单个多边形的渲染列表。

这个演示程序的主文件为 DEMOII7_1.CPP|EXE。这个程序创建一个多边形，然后绕+y 轴对其进行旋转，仅此而已。下面是这个程序中一些重要的全局变量：

```
// 初始化相机的位置和方向
POINT4D cam_pos = {0,0,-100,1};
VECTOR4D cam_dir = {0,0,0,1};

// 初始化代码
VECTOR4D vscale={.5,.5,.5,1}, vpos = {0,0,0,1}, vrot = {0,0,0,1};

CAM4DV1    cam;              // 相机
RENDERLIST4DV1 rend_list;          // 渲染列表
POLYF4DV1   poly1;            // 惟一一个多边形
POINT4D    poly1_pos = {0,0,100,1}; // 多边形在世界坐标系中的位置
```

下面是函数 Game_Init() 中的 3D 代码：

```
// 初始化数学引擎
Build_Sin_Cos_Tables();

// 初始化多边形
poly1.state = POLY4DV1_STATE_ACTIVE;
poly1.attr  = 0;
poly1.color = RGB16Bit(0,255,0);

poly1.vlist[0].x = 0;
poly1.vlist[0].y = 50;
poly1.vlist[0].z = 0;
poly1.vlist[0].w = 1;

poly1.vlist[1].x = 50;
poly1.vlist[1].y = -50;
poly1.vlist[1].z = 0;
poly1.vlist[1].w = 1;

poly1.vlist[2].x = -50;
poly1.vlist[2].y = -50;
poly1.vlist[2].z = 0;
poly1.vlist[2].w = 1;

poly1.next = poly1.prev = NULL;

// 将相机的 FOV 设置为 90 度，采用归一化投影
Init_CAM4DV1(&cam,   // 相机对象
       &cam_pos, // 相机的初始位置
       &cam_dir, // 相机的初始角度
       50.0,   // 近裁剪面和远裁剪面
       500.0
       90.0,   // 视野，单位为度
       1.0,    // 视距
       0,     // 视平面大小 - 1
       0,
       WINDOW_WIDTH,  // 视口的大小
       WINDOW_HEIGHT);
```

最后，下面函数 Game_Main() 中的重要代码，该函数对三角形执行 3D 流水线操作和渲染：

```
int Game_Main(void *parms)
{
// 这是游戏的主干，将不断被实时地调用
// 它类似于 C 语言中的 main()函数，所有调用都在这里进行

static MATRIX4X4 mrot;// 旋转矩阵
static float ang_y = 0;   // 旋转角度

int index; // 循环变量
```

```
// 启动定时时钟
Start_Clock();
// 清空可绘制面（drawing surface）
DDraw_Fill_Surface(lpddsback, 0);

// 读取键盘和其他设备输入
DInput_Read_Keyboard();

// 游戏逻辑

// 初始化渲染列表
Reset_RENDERLIST4DV1(&rend_list);

// 将多边形插入到渲染列表中
Insert_POLYF4DV1_RENDERLIST4DV1(&rend_list, &poly1);

// 创建绕 y 轴旋转的旋转矩阵
Build_XYZ_Rotation_MATRIX4X4(0, ang_y, 0, &mrot);

// 缓慢地旋转多边形
if (++ang_y >= 360.0) ang_y = 0;

// 对渲染列表中惟一一个多边形的局部坐标进行旋转变换
Transform_RENDERLIST4DV1(&rend_list, &mrot, TRANSFORM_LOCAL_ONLY);

// 执行局部坐标到世界坐标变换
Model_To_World_RENDERLIST4DV1(&rend_list, &poly1_pos);

// 创建相机变换矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// 执行世界坐标到相机坐标变换
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// 执行相机坐标到透视坐标变换
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// 执行屏幕变换
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// 锁定后缓存
DDraw_Lock_Back_Surface();

// 渲染多边形列表
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer, back_lpitch);

// 解除对后缓存的锁定
DDraw_Unlock_Back_Surface();

// 交换缓存（flip the surfaces）
DDraw_Flip();

// 同步到 30 帧/秒
Wait_Clock(30);

// 检查用户是否想退出
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);
  } // end if

// 成功返回
return(1);

} // end Game_Main
```

图 7.20 是运行 DEMOII7_1.EXE 时的屏幕截图。正如读者看到的，很有趣。这个演示程序旨在

说明，使用 10 个左右的函数调用便可以创建一个完整的 3D 线框图形。



图 7.20　DEMOII7_1.EXE 的屏幕截图

7.5.2　3D 线框立方体

- 加载的物体：一个简单立方体（CUBE1.PLG）；
- 相机类型：欧拉相机，位置固定，视野为 90 度，2×2 的归一化视平面，视口大小为 400×400；
- 投影类型：先进行透视变换，然后进行视口变换；
- 3D 流水线使用的计算方法：所有函数都使用手工计算，没有变换是基于矩阵的；
- 背面消除：没有执行；
- 物体剔除：没有执行；
- 渲染方式：渲染物体，没有使用渲染列表。

前面演示了一个旋转的多边形，现在绘制一些更有趣的东西。DEMOII7_2.CPP|EXE 与前一个演示程序类似，但使用网格加载函数加载一个立方体网格，存储该网格文件为 CUBE.PLG，如下所示：

```
# plg/plx 文件开头

# 简单立方体
tri 8 12

# 顶点列表
 5 5 5
-5 5 5
-5 5 -5
 5 5 -5
 5 -5 5
-5 -5 5
-5 -5 -5
 5 -5 -5


# 多边形列表
0xd0f0 3 2 1 0
0xd0f0 3 3 2 0
0xd0f0 3 4 7 0
0xd0f0 3 7 3 0
0xd0f0 3 6 7 4
0xd0f0 3 5 6 4
0xd0f0 3 2 6 1
0xd0f0 3 6 5 1
0xd0f0 3 7 6 3
0xd0f0 3 6 2 3
0xd0f0 3 5 4 0
0xd0f0 3 1 5 0
```

这是一个 10×10×10 的立方体。将网格加载并存储到一个 OBJECT4DV1 结构中后，在主循环中缓慢地旋转它，并在 3D 流水线中对其进行变换。然而，这里没有将立方体转换为多边形，并将它们插入到一个渲染列表中，而是将立方体保持为物体不变，以说明在整条流水线中物体可以始终为物体。

下面是这个演示程序中使用的一些重要全局变量：

```
// 初始化相机的位置和方向
POINT4D cam_pos = {0,0,0,1};
VECTOR4D cam_dir = {0,0,0,1};

// 初始化代码...
VECTOR4D vscale={5.0,5.0,5.0,1}, vpos = {0,0,0,1}, vrot = {0,0,0,1};

CAM4DV1  cam;    // 相机
```

```
OBJECT4DV1 obj;   // 用于存储立方体
```

请注意 vscale 的值，这个向量用于在加载期间对网格进行缩放，因此加载时，10×10×10 的立方体将被放到 50×50×50。当然，也可以直接对模型进行修改，但这正是加载函数的灵活性所在：使用同一个模型并在加载时对其进行变换，可以让网格细微地变形。

下面来看一下函数 Game_Init() 中的 3D 初始化代码。这与前一个演示程序中相同，但包含加载物体的代码。

```
// 将相机的 FOV 设置为 90 度，并采用归一化投影
Init_CAM4DV1(&cam,   // 相机对象
       &cam_pos, // 相机的初始位置
       &cam_dir, // 相机的初始角度
       50.0,   // 近裁剪面和远裁剪面
       500.0,
       90.0,   // 视野，单位为度
       WINDOW_WIDTH,  // 视口的大小
       WINDOW_HEIGHT);

// 加载立方体
Load_OBJECT4DV1_PLG(&obj, "cube1.plg",&vscale, &vpos, &vrot);

// 设置立方体在世界坐标系中的位置
obj.world_pos.x = 0;
obj.world_pos.y = 0;
obj.world_pos.z = 100;
```

物体被放置在离世界坐标系原点 100 单位（沿+z 轴）处，这样物体看起来将较大，投影到视平面的部分较大。最后，下面列出了函数 Game_Main() 的代码，让读者知道调用了哪些流水线处理函数。

```
int Game_Main(void *parms)
{
// 这是游戏的主干，将不断被实时地调用
// 它类似于 C 语言中的 main()函数，所有调用都在这里进行

static MATRIX4X4 mrot; // 旋转矩阵

int index; // 循环变量

// 启动定时时钟
Start_Clock();

// 清空可绘制面（drawing surface）
DDraw_Fill_Surface(lpddsback, 0);

// 读取键盘和其他设备输入
DInput_Read_Keyboard();

// 游戏逻辑

// 重置物体（仅对背面消除和物体剔除有意义）
Reset_OBJECT4DV1(&obj);

// 创建绕 y 轴旋转的旋转矩阵
Build_XYZ_Rotation_MATRIX4X4(0, 5, 0, &mrot);

// 对物体的局部坐标进行旋转变换
Transform_OBJECT4DV1(&obj, &mrot, TRANSFORM_LOCAL_ONLY,1);

// 执行局部坐标到世界坐标变换
Model_To_World_OBJECT4DV1(&obj);

// 创建相机变换矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// 执行世界坐标到相机坐标变换
World_To_Camera_OBJECT4DV1(&obj, &cam);
```

```
// 执行相机坐标到透视坐标变换
Camera_To_Perspective_OBJECT4DV1(&obj, &cam);

// 执行屏幕变换
Perspective_To_Screen_OBJECT4DV1(&obj, &cam);

// 锁定后缓存
DDraw_Lock_Back_Surface();

// 渲染物体
Draw_OBJECT4DV1_Wire16(&obj, back_buffer, back_lpitch);

// 解除对后缓存的锁定
DDraw_Unlock_Back_Surface();

// 交换缓存（flip the surfaces）
DDraw_Flip();

// 同步到 30 帧/秒
Wait_Clock(30);

// 检查用户是否想退出
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // end if

// 成功返回
return(1);

} // end Game_Main
```

同样，通过调用 10 个左右的函数，便创建了一个能够运行的 3D 演示程序。图 7.21 是运行 DEMOII7_2.EXE 时的屏幕截图。整个立方体都可见，这是由于视野较大，但物体离视点较近。



图 7.21　DEMOII7 2.EXE 的屏幕截图

7.5.3　消除了背面的 3D 线框立方体

- 加载的物体：一个简单立方体（CUBE2.PLG）；
- 相机类型：欧拉相机，位置固定，视野为 90 度，2×2/*ar* 的归一化视平面，视口大小为 640×480；
- 投影类型：先进行透视变换，然后进行视口变换；
- 3D 流水线使用的计算方法：所有函数都使用手工计算，没有变换是基于矩阵的；
- 背面消除：执行；
- 物体剔除：没有执行；
- 渲染方式：渲染物体，没有使用渲染列表。

DEMOII7_3.CPP¦EXE 与前一个演示程序类似，只是进行了背面消除，且视口更大。另外，为执行背面消除，有两件事情需要考虑。首先是模型网格本身：加载网格时，多边形可能是单面的，也可能是双面的，但仅当是单面的时，背面消除才有可能。如果是双面的，背面消除是没有意义的，因为从两边都能够看到它。在线框模式下，背面消除显得并不那么重要，但只要愿意仍可以在网格数据中启用背面消除。需要做的第二件事情是，必须在局部坐标到世界坐标变换之后和相机变换之前进行背面测试。

首先，来看一看网格数据 CUBE1.PLG 中是否启用了背面消除。下面是该文件中的一个多边形描述符行：

```
# 多边形列表
0xd0f0 3 2 1 0 # 多边形 0
```

其中第一个数字是多边形描述符。本章前面讨论加载.PLG/PLX 文件的一节介绍过，在 16 位的多边形描述符中，D 位指定了多边形是否是双面的：

```
CSSD ¦ RRRR¦ GGGG ¦ BBBB
```

换句话说，D 位是第一个十六进制位中最后一个二进制位。在上述多边形描述符中，第一个十六进制位为 0xD，对应的二进制数为 1101，因此多边形是双面的。必须将多边形设置为单面的，为此，创建了另一个名为 CUBE2.PLG 的文件，附带光盘中包含这个文件。在这个文件中，D 为被设置为 0，因此多边形描述符的前 4 位为 1100（0xC）。对于上述多边形描述符，将被修改成这样：

```
# 多边形列表
0xc0f0 3 2 1 0 # 多边形 0
```

为删除背面，需要调用下述函数：

```
// 背面消除
Remove_Backfaces_OBJECT4DV1(&obj, &cam);
```

上述函数调用是在根据观察角度创建相机变换矩阵之后和执行世界坐标到相机坐标变换之前进行的，这样不可见的物体将不被执行世界坐标到相机坐标变换。

图 7.22 是运行 DEMOII7_3.EXE 时的屏幕截图。用户通过按上箭头和下箭头键，可以使立方体绕 x 轴旋转。

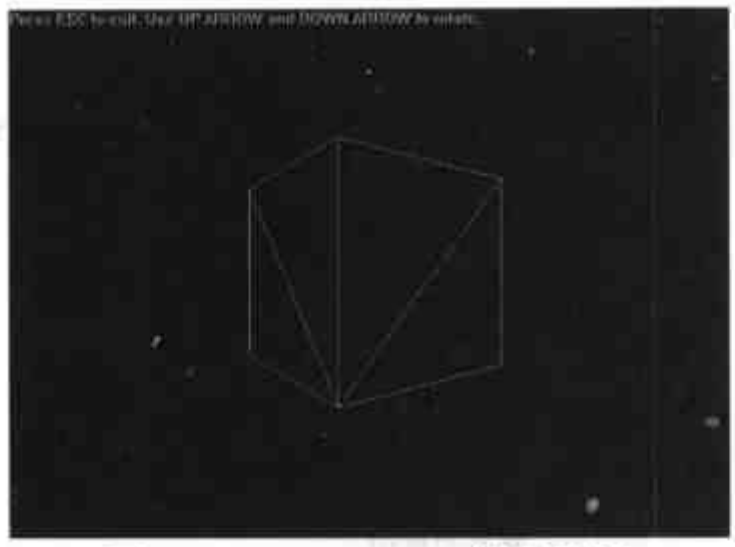


图 7.22 DEMOII7_3.EXE 的屏幕截图

7.5.4 3D 坦克演示程序

- 加载的物体——一辆简单坦克（TANK1.PLG）。
- 相机类型——可旋转的欧拉相机，视野为 90 度，2×2 的归一化视平面，视口大小为 400×400。
- 投影类型——先进行透视变换，然后进行视口变换。
- 3D 流水线使用的计算方法——所有函数都使用手工计算，没有变换是基于矩阵的。
- 背面消除——没有执行。
- 物体剔除——执行。
- 渲染方式——包含多个物体的渲染列表。



图 7.23 DEMOII7_4.EXE 的屏幕截图

图 7.23 是运行 DEMOII7_4.EXE 的屏幕截图，这种截图不好捕获，但其中还是有几辆坦克。这是一个非常复杂的演示程序，首先按通常那样设置相机，然后加载坦克，并进入主循环。

在这个演示程序中，有趣的是如何创建多个坦克拷贝：这里没有加载多辆坦克，而是通过多次修改坦克的位置，并将其插入到主渲染列表中。这样，位于世界空间中特定位置的坦克的多边形将被变换，并插入到渲染列表中，使得看起来有多辆坦克。对渲染列表执行世界坐标到相机坐标变换和透视变换时，将有多辆坦克。下面是该程序的主循环中的重要代码，其中复制坦克并将其多边形插入到渲染列表中的代码为粗体：

```
int Game_Main(void *parms)
{
// 这是游戏的主干，将不断被实时地调用
// 它类似于 C 语言中的 main()函数，所有调用都在这里进行

static MATRIX4X4 mrot; // 旋转矩阵
static float x_ang = 0, y_ang = 5, z_ang = 0;
char work_string[256];

int index; // 循环变量

// 启动定时时钟
Start_Clock();

// 清空可绘制面（drawing surface）
DDraw_Fill_Surface(lpddsback, 0);
```

```
// 读取键盘和其他设备输入
DInput_Read_Keyboard();

// 游戏逻辑

// 重置渲染列表
Reset_RENDERLIST4DV1(&rend_list);

// 重置角度
x_ang = 0;
y_ang = 1;
z_ang = 0;

// 判断用户是否想旋转相机
if (KEY_DOWN(VK_DOWN))
  cam.dir.x+=1;
else
if (KEY_DOWN(VK_UP))
  cam.dir.x-=1;

// 判断用户是否想旋转相机
if (KEY_DOWN(VK_RIGHT))
  cam.dir.y-=1;
else
if (KEY_DOWN(VK_LEFT))
  cam.dir.y+=1;

// 创建旋转矩阵
Build_XYZ_Rotation_MATRIX4X4(x_ang, y_ang, z_ang, &mrot);
// 对物体的局部坐标进行旋转变换
Transform_OBJECT4DV1(&obj, &mrot, TRANSFORM_LOCAL_ONLY,1);

// 执行物体剔除
strcpy(buffer,"Objects Culled: ");

for (int x=-NUM_OBJECTS/2; x < NUM_OBJECTS/2; x++)
  for (int z=-NUM_OBJECTS/2; z < NUM_OBJECTS/2; z++)
  {
  // 重置物体（仅对背面消除和物体剔除有意义）
  Reset_OBJECT4DV1(&obj);

  // 设置物体的位置
  obj.world_pos.x = x*OBJECT_SPACING+OBJECT_SPACING/2;
  obj.world_pos.y = 0;
  obj.world_pos.z = 500+z*OBJECT_SPACING+OBJECT_SPACING/2;

  // 执行物体剔除操作
  if (!Cull_OBJECT4DV1(&obj, &cam, CULL_OBJECT_XYZ_PLANES))
    {
    // 物体是可见的
    // 可以将其插入到渲染列表中
    // 执行局部/模型坐标到世界坐标变换
    Model_To_World_OBJECT4DV1(&obj);

    // 将物体插入到渲染列表中
    Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj);
    } // end if
  else
    {
    sprintf(work_string, " [%d, %d] ", x,z);
    strcat(buffer, work_string);
    }

  } // end for

Draw_Text_GDI(buffer, 0, WINDOW_HEIGHT-20, RGB(0,255,0), lpddsback);
```

```
// 创建相机变换矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// 背面消除
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);
// 执行世界坐标到相机坐标变换
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// 执行相机坐标到透视坐标变换
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// 执行屏幕变换
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// 显示程序用法说明
Draw_Text_GDI("Press ESC to exit. Use ARROW keys to rotate camera.",
       0, 0, RGB(0,255,0), lpddsback);

// 锁定后缓存
DDraw_Lock_Back_Surface();

// 渲染物体
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer, back_lpitch);

// 解除对后缓存的锁定
DDraw_Unlock_Back_Surface();

// 交换缓存（flip the surfaces）
DDraw_Flip();

// 同步到 30 帧/秒
Wait_Clock(30);

// 检查用户是否想退出
if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
  {
  PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // end if

// 成功返回
return(1);

} // end Game_Main
```

现在读者应该知道渲染列表有多重要吧，与分别渲染各个物体相比，使用渲染列表是一种更佳的渲染方法。对于进行变换、动画、AI、逻辑等而言，使用物体是不错的，但进入渲染阶段后，最好（在执行物体剔除后）将物体分解成多边形，并将这些多边形插入到一个主渲染列表中。

7.5.5 相机移动的 3D 坦克演示程序

- 加载的物体——一辆简单坦克（TANK1.PLG）；
- 相机类型——可旋转的 UVN 相机，视野为 90 度，2×2*/ar* 的归一化视平面，视口大小为 800×600，全屏；
- 投影类型——先进行透视变换，然后进行视口变换；
- 3D 流水线使用的计算方法——所有函数都使用手工计算，没有变换是基于矩阵的；
- 背面消除——没有执行；
- 物体剔除——执行；
- 渲染方式——包含多个物体的渲染列表。

图 7.24 是 DEMOII7_5.EXE 的屏幕截图。这是已介绍过的演示程序中最复杂的，它类似于“克隆人的进攻”。这个演示程序使用 UVN 相机模型，它与其他演示程序之间的惟一差别在于，它调用

函数 Build_CAM4DV1_Matrix_UVN() 而不是 Build_CAM4DV1_Matrix_Euler() 来创建相机变换矩阵。另外，使用了参数 UVN_MODE_SIMPLE，它告诉相机变换矩阵创建函数，我们只手工提供目标位置。

因此，该函数将只计算向量 **u**、**v** 和 **n**，而不执行其他计算。函数 Game_Main() 与前一个演示程序中极其相似，但包含一项有趣的内容：计算 UVN 相机模型的相机位置。下面是计算相机位置的代码：

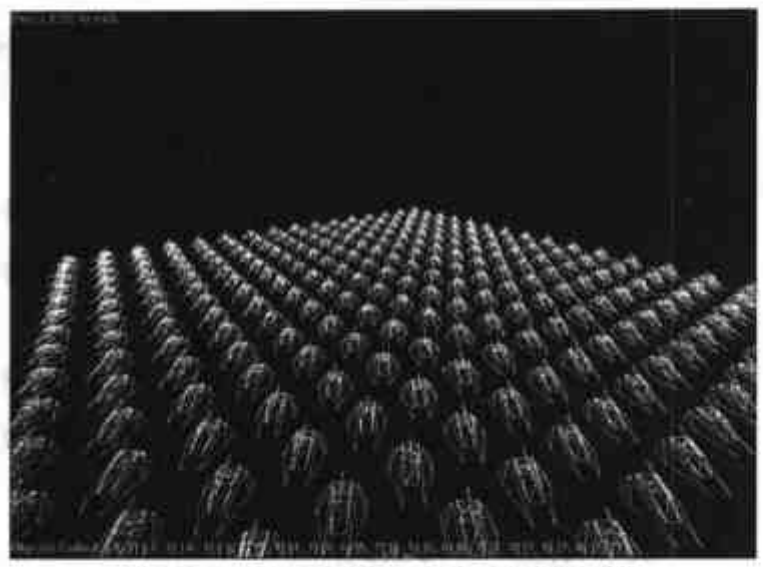


图 7.24　DEMOII7_5.EXE 的屏幕截图

```
// 计算相机的新位置，相机绕圆周运动
cam.pos.x = camera_distance*Fast_Cos(view_angle);
cam.pos.y = camera_distance*Fast_Sin(view_angle);
cam.pos.z = 2*camera_distance*Fast_Sin(view_angle);

// 增加环绕角度
if ((view_angle+=1)>=360)
   view_angle = 0;

// 生成相机变换矩阵
Build_CAM4DV1_Matrix_UVN(&cam, UVN_MODE_SIMPLE);
```

图 7.25 说明了游戏世界中物体的排列情况。这是一个使用简单数学计算实现有趣的相机移动的典范。在这个演示程序中，相机沿圆周（也可能是椭圆，这取决于倍增因子）移动，同时其高度（相对于 *x-z* 平面）呈正弦波变化，这是一种很不错的相机移动轨迹。之所以能够这样移动相机，是因为使用的是 UVN 模型。在这个演示程序中，通过指定注视点和相机位置来设置相机，UVN 模型刚好能够满足这样的要求，它提供了我们所需的控制方式。相机被放在环绕目标的圆周上，而目标位于坦克群的中心。

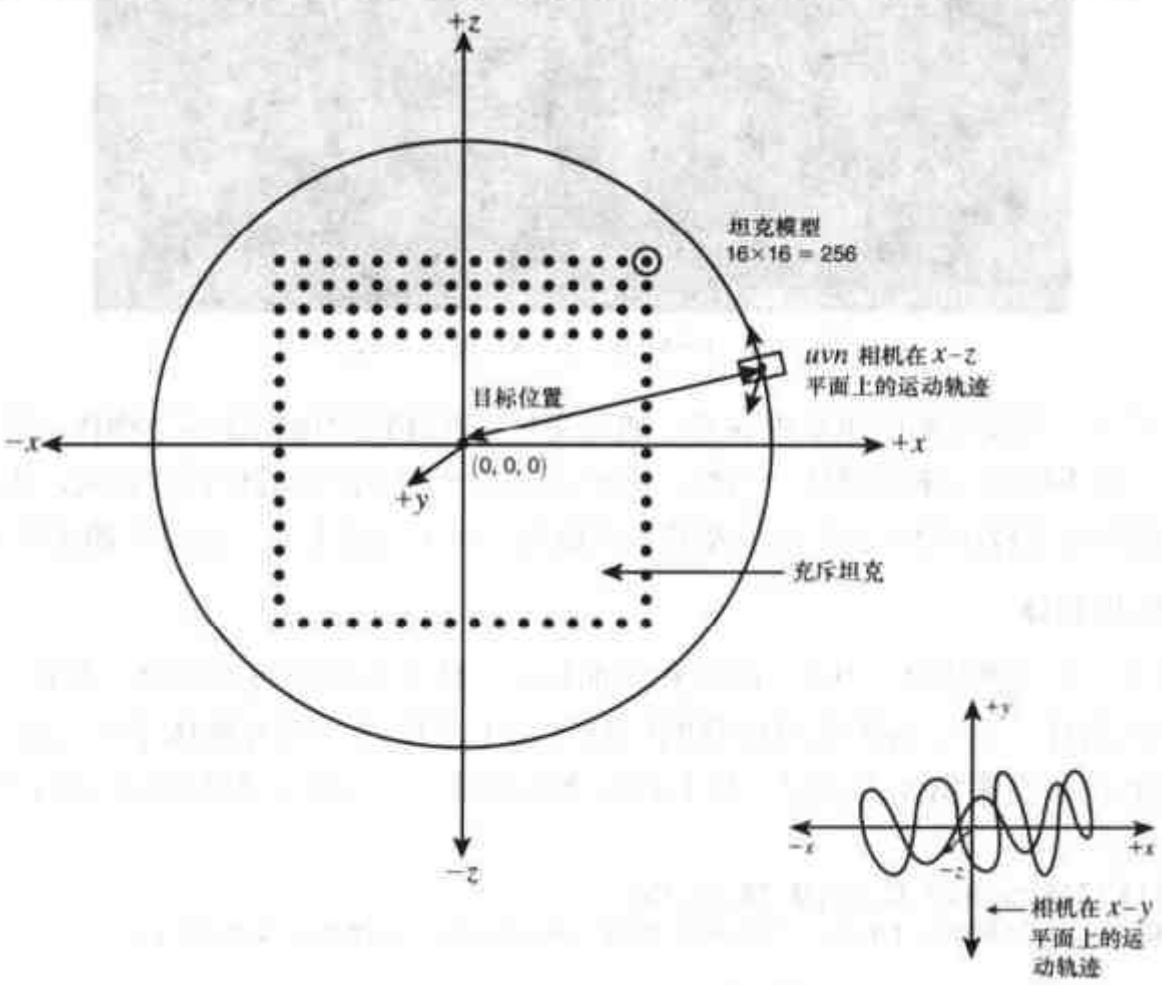


图 7.25　游戏世界中物体的排列情况

这种相机模型非常适合用于从第三者的角度观察场景。也就是说，在需要注视某个点时，可以使用 UVN 相机模型，并根据相机位置和目标位置计算 UVN 向量，进而计算旋转矩阵。然而，对于第一人称游戏（通过角色的眼睛来观察场景），基于球面坐标的 UVN 模型（即指定相机位置以及仰角和方位角）更合适。最

后一个演示程序将使用这种技术：在 UVN 相机模型中使用球面坐标系。

7.5.6 战区漫步演示程序

- 加载的物体——坦克（TANK1.PLG）和高塔（TOWER1.PLG）；
- 相机类型——有方位角、可旋转的欧拉相机，视野为 120 度，2×2/*ar* 的归一化视平面，视口大小为 800×600，全屏；
- 投影类型——先进行透视变换，然后进行视口变换；
- 3D 流水线使用的计算方法——所有函数都使用手工计算，没有变换是基于矩阵的；
- 背面消除——没有执行；
- 物体剔除——执行；
- 渲染方式——包含多个物体的渲染列表。

这是至今为止最酷的演示程序。它融合了我们介绍过的所有内容，使用了一些游戏编程技巧。这个演示程序中包含三类物体：高塔、坦克和地面标记。由于使用的是线框渲染模式，难以实现深度感，因此在游戏空间中有规则地放置了大量的小型棱锥，以实现地平面假象。另外，还使用两个颜色较深的矩形来创建了天空和地面，如图 7.26 所示。

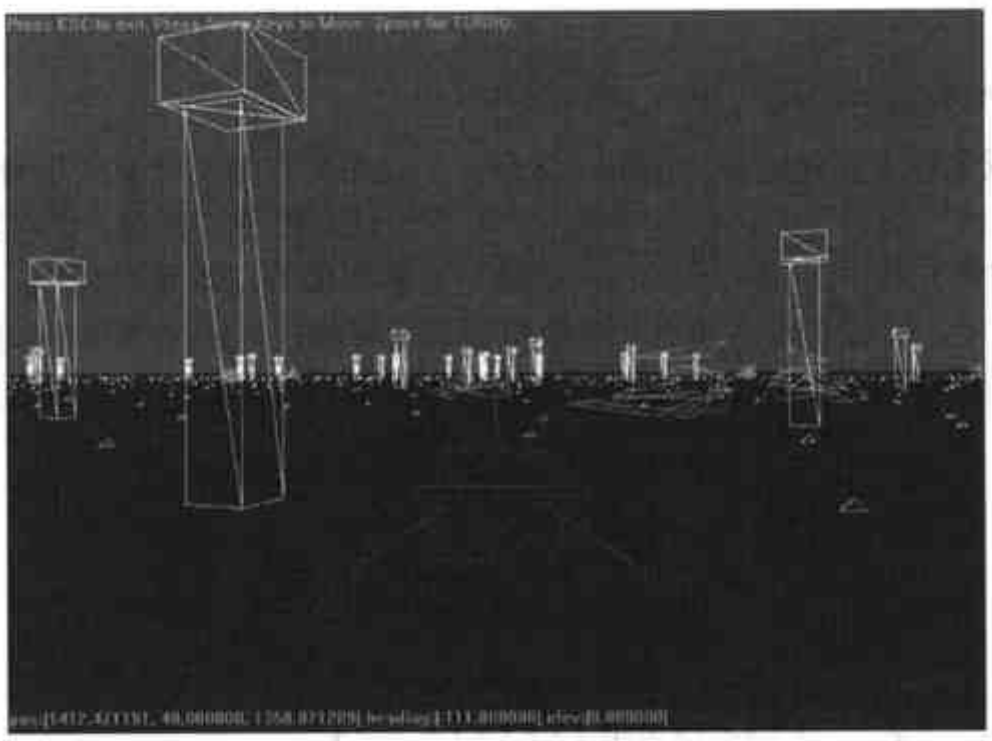


图 7.26 DEMOII7_6.EXE 的屏幕截图

对于这个演示程序，需要说明的重要内容是，通过变换主物体轻松地创建多个物体，然后将变换后的物体插入到渲染列表中，而不破坏原来的物体。当然，需要记录各个物体拷贝的位置和朝向，为此只需使用两个向量数组即可。下面描述演示程序中的各个部分及其工作原理。由于代码太多，这里只摘录其中的一些。

1. 加载和初始化物体

正如前面指出的，有三类物体：坦克、高塔和地面标记。对于其中的每类物体，都有一个主物体（master object），它是最初的物体。各个物体拷贝特有的信息（这里为位置，因为物体是静止的）被存储在辅助数据结构中。下面是加载和设置物体的代码。其中对每个网格进行了缩放，有时候必须做类似于这样的调整。

```
// 加载主坦克拷贝
VECTOR4D_INITXYZ(&vscale,0.75,0.75,0.75);
Load_OBJECT4DV1_PLG(&obj_tank, "tank2.plg",&vscale, &vpos, &vrot);

// 加载玩家的坦克模型，并从第三人称的角度观察它
VECTOR4D_INITXYZ(&vscale,0.75,0.75,0.75);
Load_OBJECT4DV1_PLG(&obj_player, "tank3.plg",&vscale, &vpos, &vrot);

// 加载主高塔拷贝
VECTOR4D_INITXYZ(&vscale,1.0, 2.0, 1.0);
```

```
Load_OBJECT4DV1_PLG(&obj_tower, "tower1.plg",&vscale, &vpos, &vrot);

// 加载主地面标记拷贝
VECTOR4D_INITXYZ(&vscale,3.0,3.0,3.0);
Load_OBJECT4DV1_PLG(&obj_marker, "marker1.plg",&vscale, &vpos, &vrot);

// 放置坦克
for (index = 0; index < NUM_TANKS; index++)
  {
  // 随机放置坦克
  tanks[index].x = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  tanks[index].y = 0; // obj_tank.max_radius;
  tanks[index].z = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  tanks[index].w = RAND_RANGE(0,360);
  } // end for


// 放置高塔
for (index = 0; index < NUM_TOWERS; index++)
  {
  // 随机放置高塔
  towers[index].x = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  towers[index].y = 0; // obj_tower.max_radius;
  towers[index].z = RAND_RANGE(-UNIVERSE_RADIUS, UNIVERSE_RADIUS);
  } // end for
```

2. 相机系统

相机非常简单。使用的是欧拉模型，相机在游戏空间中移动，并绕 y 轴旋转。也可以修改相机相对于 x 轴的角度或仰角，但由于地平面是人造的，当仰角发现变化时，必须计算人造地平面矩形的哪些地方将发生变化，这非常麻烦。下面是移动相机的代码。请注意其中的变量 turning，它用于实现一种效果。

```
// 允许用户移动相机

//提高速度
if (keyboard_state[DIK_SPACE])
   tank_speed = 5*TANK_SPEED;
else
   tank_speed = TANK_SPEED;

// 前进/后退
if (keyboard_state[DIK_UP])
   {
   // 前进
   cam.pos.x += tank_speed*Fast_Sin(cam.dir.y);
   cam.pos.z += tank_speed*Fast_Cos(cam.dir.y);
   } // end if

if (keyboard_state[DIK_DOWN])
   {
   // 后退
   cam.pos.x -= tank_speed*Fast_Sin(cam.dir.y);
   cam.pos.z -= tank_speed*Fast_Cos(cam.dir.y);
   } // end if

// 旋转
if (keyboard_state[DIK_RIGHT])
   {
   cam.dir.y+=3;

   // 让坦克稍微转向
   if ((turning+=2) > 15)
    turning=15;

   } // end if

if (keyboard_state[DIK_LEFT])
```

```
    {
    cam.dir.y-=3;

    // 让坦克稍微转向
    if ((turning-=2) < -15)
     turning=-15;

    } // end if
  else // 重新摆直
    {
    if (turning > 0)
      turning-=1;
    else
    if (turning < 0)
      turning+=1;

    } // end else

  // 创建相机变换矩阵
  Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
```

上述代码很简单。根据用户按下的键，相机位于沿一个向量移动，该向量取决于当前的方位角（与 y 轴之间的夹角）。由于位于 x-z 平面上，因此移动相机的代码如下：

```
cam.pos.x += tank_speed*Fast_Sin(cam.dir.y);
cam.pos.z += tank_speed*Fast_Cos(cam.dir.y);
```

另外，相机旋转只不过是更新变量 cam.dir.y 而已，因为上述代码清单的最后函数调用使用它来创建相机变换矩阵。比较有趣的是变量 turning 及其用途。如果读者运行该演示程序，将看到屏幕的最前面有一辆坦克，这表示玩家的位置。如果玩家按下左或右箭头键，这辆坦克将缓慢地改变方向，玩家松开这些键后，坦克将回到原来的方向。这是通过使用变量 turning 来跟踪转向状态实现的。玩家转向时，总角度被限制在某个值以下（这里为 15 度），达到这个值后，便不能进一步转向。玩家停止转向后，坦克将重新回到原来的方向。这是一种非常酷的效果，只需要几行代码就可实现，很多第三人称赛车游戏都使用。当然，在基于物理学的模拟中，这项工作是由物理模型完成的，但就我们的目的而言，现在这样做是可行的。

3. 生成游戏世界

游戏世界（arena）由大量静态物体组成。这些物体是通过复制主物体得到的，它们被放置在游戏世界中固定的位置上。各个物体是通过重新指定主物体的位置，对其进行变换并插入到主多边形渲染列表中（执行了物体剔除）来创建的。这种方法的优点在于，原来的物体没有损坏。由于只使用变换后的顶点，因此原来的模型毫发未损。在物体数据结构中，包含两个存储顶点数据的数组。下面是将所有的高塔模型加入到游戏世界中的代码：

```
// 将高塔加入到游戏世界中
for (index = 0; index < NUM_TOWERS; index++)
  {
  // 重置物体（仅对背面消除和物体剔除有意义）
  Reset_OBJECT4DV1(&obj_tower);
// 设置高塔的位置
obj_tower.world_pos.x = towers[index].x;
obj_tower.world_pos.y = towers[index].y;
obj_tower.world_pos.z = towers[index].z;

// 执行物体剔除操作
if (!Cull_OBJECT4DV1(&obj_tower, &cam, CULL_OBJECT_XYZ_PLANES))
  {
  // 物体是可见的
  // 可以将其插入到渲染列表中
  // 执行局部/模型坐标到世界坐标变换
  Model_To_World_OBJECT4DV1(&obj_tower);

  //将物体插入到渲染列表中
  Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj_tower);
```

```
    } // end if

} // end for
```

是不是很简单？最后来看看对玩家的坦克模型的处理方式。

4. 渲染玩家的坦克模型

当前，我们的引擎并不太先进，无法将相机固定在物体上，并随物体一起移动。我们必须手工实现这种效果。在这个演示程序中，相机的位置是已知的，因此可以基于相机的方位角来完成这项工作。将坦克模型放在相机前面较合适的位置（约 400 个单位），并使其高度位置比相机稍低。然后，根据当前的观察角度旋转该坦克模型（该模型不同于物体放置函数使用的坦克模型），然后对局部坐标到世界坐标变换，这样便得到了一辆表示玩家的坦克。

完成上述任务的代码很短：

```
// 将玩家的坦克加入到游戏世界中
// 重置物体（仅对背面消除和物体剔除有意义）
Reset_OBJECT4DV1(&obj_player);

// 设置坦克的位置
obj_player.world_pos.x = cam.pos.x+400*Fast_Sin(cam.dir.y);
obj_player.world_pos.y = cam.pos.y-50;
obj_player.world_pos.z = cam.pos.z+400*Fast_Cos(cam.dir.y);

// 生成绕 y 轴旋转的旋转矩阵
Build_XYZ_Rotation_MATRIX4X4(0, cam.dir.y+turning, 0, &mrot);
// 对物体的局部坐标执行旋转变换
Transform_OBJECT4DV1(&obj_player, &mrot, TRANSFORM_LOCAL_TO_TRANS,1);

// 执行局部坐标到世界坐标变换
Model_To_World_OBJECT4DV1(&obj_player, TRANSFORM_TRANS_ONLY);

// 将物体插入到渲染列表中
Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj_player);
```

请注意其中的粗体代码，这行代码在当前方位角的基础上加上一个很小的转向角度，以实现转向效果。

注意：虽然在大多数演示程序中，使用的是 16 位的颜色；但不幸的是，使用软件光栅化和 16 位颜色来绘制着色和纹理映射图形时，难以得到合理的填充速度。为提高速度和演示调色板颜色模式下的优化方法，很多工作都将在 8 位颜色模式下来完成。

7.6 总　　结

本章介绍的内容很多，同时编写了大量的代码，作者的手指都快磨破了，但这是值得的。读者可以根据自己编写 3D 引擎的需要学习相应的内容，但作者希望知道所有细节和完成工作的各种方法，但愿读者也是如此。本章介绍了大量的基本知识，如 3D 数据结构、从磁盘中加载物体、局部坐标到世界坐标变换、世界坐标到相机坐标变换、相机坐标到透视坐标变换、屏幕变换、物体剔除、背面消除、何时使用 4D 坐标、为何使用矩阵并非总是最佳的变换方式、以线框模式渲染物体、UVN 相机模型以及大量的演示程序。下一章将使用着色技术和光照来提高 3D 物体的真实感；处理 3D 裁剪，这项工作不能再往后推了；还需要编写一个更好的模型加载函数。

第三部分

基本 3D 渲染

第 8 章：基本光照和实体造型

第 9 章：插值着色技术和仿射纹理映射

第 10 章：3D 裁剪

第 11 章：深度缓存和可见性

第 8 章　基本光照和实体造型

本章通过将物体渲染为实体（solid）而不是线框，来增强 3D 场景的真实感。这意味着将涉及着色和光照；另外，与将物体渲染为线框相比，背面剔除的影响将大得多，因为现在无法看到物体后面的东西。本章将概要地讨论光照和着色，并分别使用 8 位颜色索引模式和 16 位 RGB 颜色模式实现涉及的每项内容。另外，还将对 PLG/PLX 文件加载器进行改进，使之支持其他的文件格式，从而避免手工创建模型。本章既介绍理论，也涉及到实践，虽然其中的大部分理论将到后续章节中才得到应用，但现在正是介绍它们的好时机。本章包含以下主题：

- 基本光照模型；
- 环境光；
- 散射光；
- 镜面反射光；
- 定向光源；
- 点光源；
- 聚光灯；
- 三角形的光栅化；
- 光照时对位深的考虑；
- 固定（constant）着色；
- 恒定（flat）着色；
- Gouraud 着色；
- Phong 着色；
- 深度排序；
- 加载模型。

8.1　计算机图形学的基本光照模型

光照是计算机图形学中的一个庞大主题，它研究的是光子，或多或少地与电子相互作用有关。光子是一种能量，同时具有粒子性和波动性。光子具有位置和速度（粒子性），还有频率（波动性）。图 8.1 的光谱图描述了红外线、可见光、紫外线、X 射线等的频率范围。当然，所有的光都以光速传播，这种速度用 c 表示，其值为 3.0×10^8 米/秒。

频率（F，单位为赫兹）与波长（λ，单位为米）之间的关系如下：

$$F = c / \lambda$$

例如，红色光的波长约 700nm（纳米，10^{-9} 米），其频率如下：

$$F = 3.0\times10^{8}\text{m/s} / (700\times10^{-9})\ \text{m} = 4.286 \times 10^{14}\ \text{Hz}$$

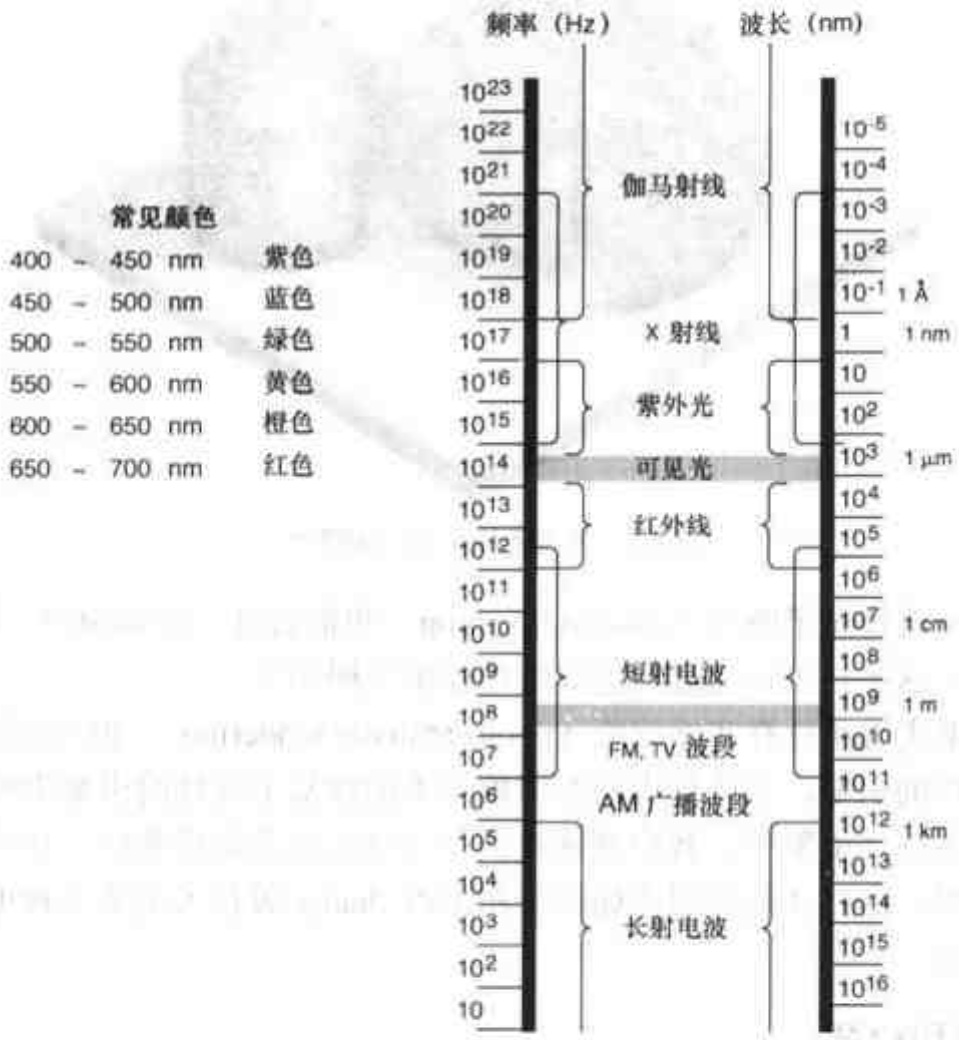


图 8.1　电磁光谱

读者可能认为，光速已经非常高，但相对于整个宇宙空间而言，这种速度太慢了。然而，光速是速度的极限值：光的速度是不能提高的，要提高必须提供无穷大的能量，有关这方面的内容不在本书的探讨范围之内。

光线的颜色取决于频率（波长）。那么，光强又是什么呢？它与每秒发射的光子数相关。亮度是每秒钟照射到表面上的光子数，因此表面 S 每秒吸收的总能量为：

每秒吸收的能量 $= e_{photon}(\lambda)$*光子数/秒

其中光子的能量是频率的函数。

当然，大多数 3D 图形引擎不依赖于对光子或物质的使用，而是使用模型。几乎所有的 3D 渲染软件包都使用光线跟踪：跟踪从视点发出的、穿过场景的光线，确定光线与场景中几何体的交点，并据此对屏幕上的像素进行着色。图 8.2 通过三个简单物体说明了这种过程，图 8.3 是一个使用光线跟踪绘制的场景（请注意其中的阴影、反射和亮点）。

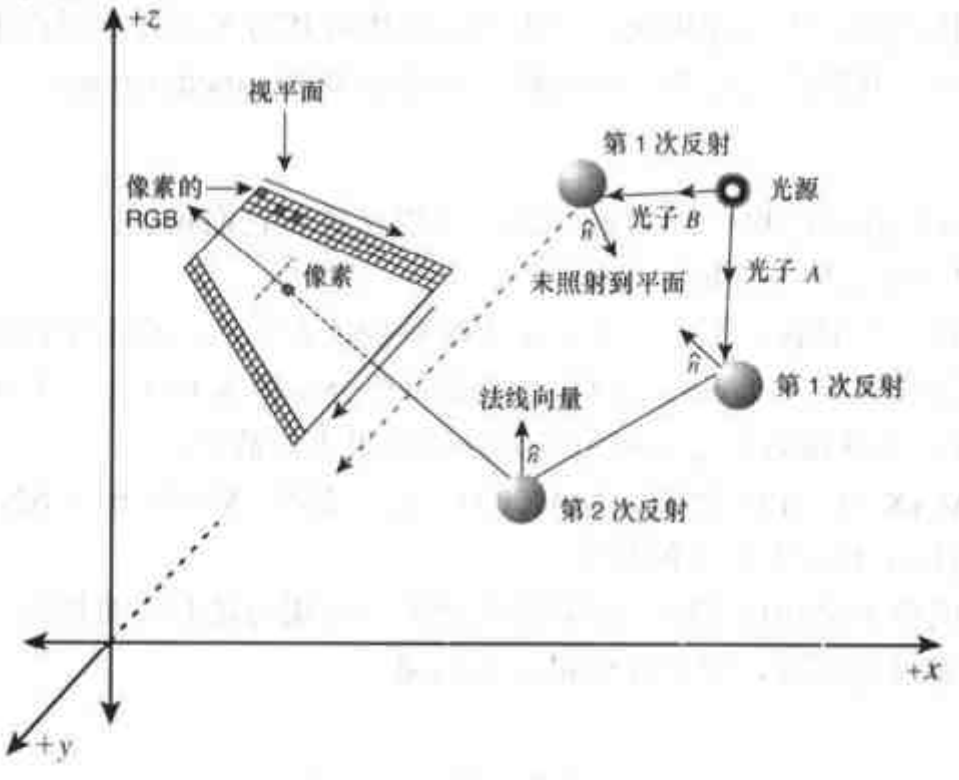


图 8.2　光线跟踪物理学



图 8.3　使用光线跟踪绘制的场景

另外，读者肯定还听说过光子映射（photon mapping）和辐射度（radiosity）等技术，它们以更复杂的方式来模拟光子的影响。这些技术旨在尽可能真实地模拟光照场景。

虽然我们想尽可能地实现照片真实感渲染（photo-realistic rendering），但这受限于使用的软件和硬件。我们要创建的是基于软件的引擎，因此使用的光照模型不能像基于硬件的引擎中那么复杂。然而，这正是有趣的地方之一：看看在软件引擎中，我们能够在执行实时光照方面达到什么样的程度。有鉴于此，我们必须极其深入地了解光照，这样才能使用诸如像素和顶点 shader 等技术来更大程度地挖掘出硬件的潜力，因为我们知道问题的真相。

8.1.1　颜色模型和材质

物体的材质决定了它将如何与光线交互。例如白色光（包含所有的可见光）照射到表面时，表面的很多属性决定了光线的行为。有些光被反射，被反射的光的颜色决定了物体的颜色；而其他光被表面吸收。另外，物体可能并非完全不透明的，在这种情况下，有些光将穿透物体（折射）。最后，光子照射到材质上时，可能导致它发射光（荧光或磷光）。所有这些属性都可能与波长或强度相关。因此，模拟材质和光线之间的交互涉及的工作量非常大。通常，使用为数不多的几个参数来模拟材质，如反射率、色散、镜面反射率和发射率等。

正如读者知道的，计算机屏幕上的颜色是使用 RGB 或颜色索引（8 位颜色）来表示的。但这里重点介绍 RGB 颜色模式，在这种模式下，每个分量被表示为 0～1 或 0～255 的值。例如，在 24 位的 RGB 模式下，每个颜色分量用 8 位表示，因此红色的 RGB 值为（255, 0, 0）。

使用浮点数格式（即每个分量用 0～1 之间的值表示）时，红色的 RGB 值为(1.0，0.0，0.0)。这两种 RGB 值表示的是同一种颜色。然而，对数学运算而言，浮点数格式比字节值（byte-valued）格式更容易处理。另一方面，字节值格式的速度可能更快，因为可以直接将其写入到视频缓存中，而无需进行转换。下面介绍使用这两种格式时，对颜色执行的两种运算：加法和调制（modulation）。

1．颜色加法

给定 RGB 颜色 C1(r1, g1, b1) 和 C2(r2, g2, b2)，可以这样将它们相加：

$C_{sum} = C1 + C2 = (r1 + r2, g1 + g2, b1 + b2)$

然而，这种运算存在一个问题：溢出。无论采用哪种颜色表示法，将两个值相加后，如果结果大于分量的最大值，必须对其进行截取（clamp）。例如，假设每个分量用 8 位表示，其取值范围为 0 到 255，则将颜色 C1 和 C2 相加后，必须按如下方式对每个分量的结果进行截取：

$C_{sum} = C1 + C2 = (\mathrm{MAX}(r1 + r2, 255), \mathrm{MAX}(g1 + g2, 255), \mathrm{MAX}(b1 + b2, 255))$

其中 MAX(x，y) 返回 x 和 y 中较大的那个。

这样便可确保分量的值不会超过 255。分量超过 255 的后果可能是灾难性的，因为这样的值将导致溢出，在大多数情况下，将进行回绕，得到完全错误的结果。

2．颜色调制

颜色调制实际上就是颜色乘法。先来看一个简单的例子——将颜色 C1 乘以标量 s：

$C_{modulated}$ = s*C1 = （s*r1, s*g1, s*b1）

其中 s 可以是 0 到无穷大的任何值。缩放因子为 1.0 时，颜色保持不变；缩放因子为 2.0 时，颜色的亮度将增加 1 倍；缩放因子为 0.1 时，颜色的亮度将降低到原来的 1/10。同样，这里也需要避免溢出——将结果截取到 1.0 或 255，具体截取到哪个值取决于颜色表示法。有关使用统一的缩放因子对颜色进行调制的例子，请参阅 DEMOII8_1.CPP|EXE。通过按上、下箭头键，可以调制纹理像素，从而提高或降低显示的纹理的亮度。另外，还可以通过按左、右箭头键来更换纹理。注意，如果亮度的调整程度过大，纹理的 RGB 分量将到达极限值，导致颜色发生变化。解决这种问题的方法之一是，在调制阶段检查是否有某个分量达到最大值（如 255），如果是这样，则让像素的颜色保持不变。如果在某个分量达到最大值时增大其他分量的值，将改变像素的颜色，因为不能保持分量之间的比值不变。

上述调制方式实际上只是调整了亮度；另一种调制方式是 RGB 乘法，它将两种颜色相乘，即用颜色 C1 调制颜色 C2 的结果如下：

$C_{modulated}$ = C1*C2 = （r1*r2, g1*g2, b1*b2）

现在，颜色乘法要复杂些，我们必须就乘法的含义达成一致。例如，如果 C1 为 RGB 格式（0...1，0...1，0...1），则 C2 的格式将无关紧要；结果将不会溢出，C1 只是调制了 C2 的强度。这正是光映射（light mapping）的工作原理。对于名为 LIGHT_TEXTURE[x][y]的光照纹理（light texture），其每个像素都被解释为 0 到 1 的光值，并将其与颜色纹理 COLOR_TEXTURE[x][y]相乘，结果如图 8.4 所示。在该图中，有一个聚光灯纹理，它表示一个光照纹理，我们将使用它来调制颜色纹理（注意，这个光照纹理是单色的），图中还显示了调制结果。这种调制存在的惟一问题是，结果并非总是像您预期的那样，因此可能需要使用一种更复杂的调制流水线。例如，可能采用下列两种方式：

```
pixel_dest[x,y]rgb = pixel_source[x,y]rgb *ambient +
pixel_source[x,y]rgb * light_map[x,y]rgb
pixel_dest[x,y]rgb = pixel_source[x,y]rgb+ambient *
light_map[x,y]rgb
```

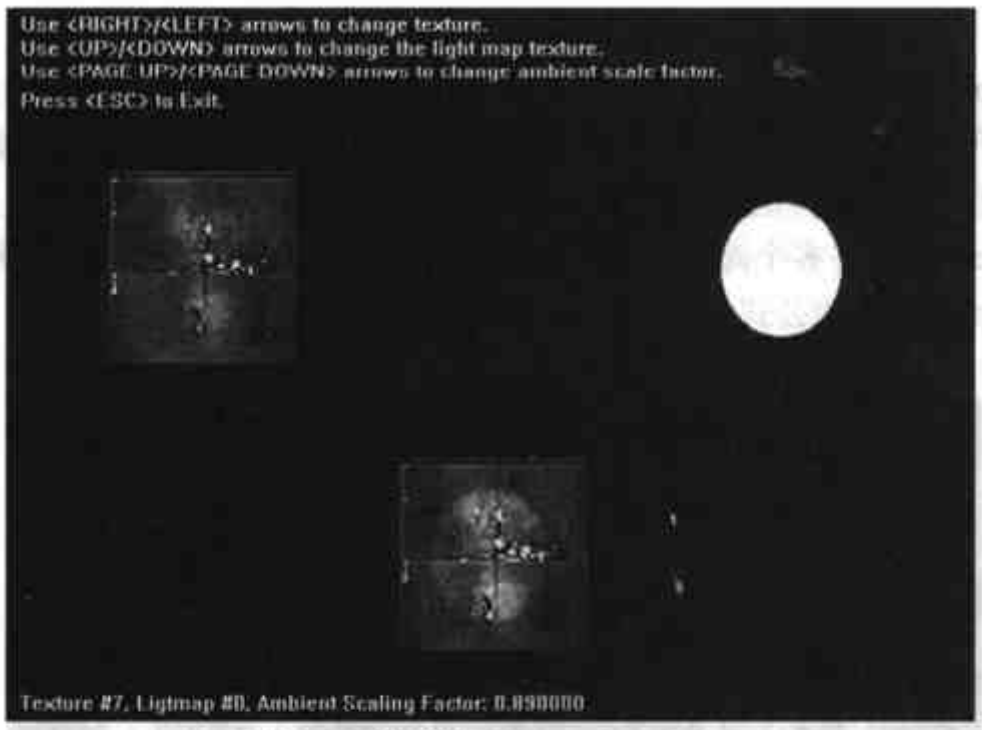


图 8.4　光调制（light modulation）

注意：[x，y]表示图像中所有的像素，下标 rgb 表示对每个分量（R、G 和 B）分别进行处理。

其中第一个公式的含义是，对于纹理图中的每个像素颜色进行缩放，并使用光照纹理对纹理图进行调制，然后将上述两种操作的结果相加。正如读者看到的，结果比仅进行调制要亮一些。从本质上说，这相当于将调制结果和一个基纹理（base texture）相加。ambient 是一个标量，决定了纹理在结果中的权重，即纹理对输出的影响程度。另外，第二个公式显得更合适，它首先修改光照纹理图的亮度，然后将其与源纹理图相加。

有关使用光映射和调制的范例，请参阅 DEMOII8_2.CPP|EXE（它使用第一个公式）和 DEMOII8_3.CPP|EXE。这两个演示程序分别使用前面的两个公式，用一个聚光灯纹理来调制一个墙壁纹理。请注意这两个演示程序之间的差别，前者采用颜色调制，因此仅当基纹理中包含绿色分量时，才能看到彩

色光；而后者使用颜色加法，因此总能看到绿色光（如图 8.5 所示）。这两种方法都很有用，具体采用哪种方法，取决于是否要求逼真。读者可以使用箭头键来控制这两个演示程序。

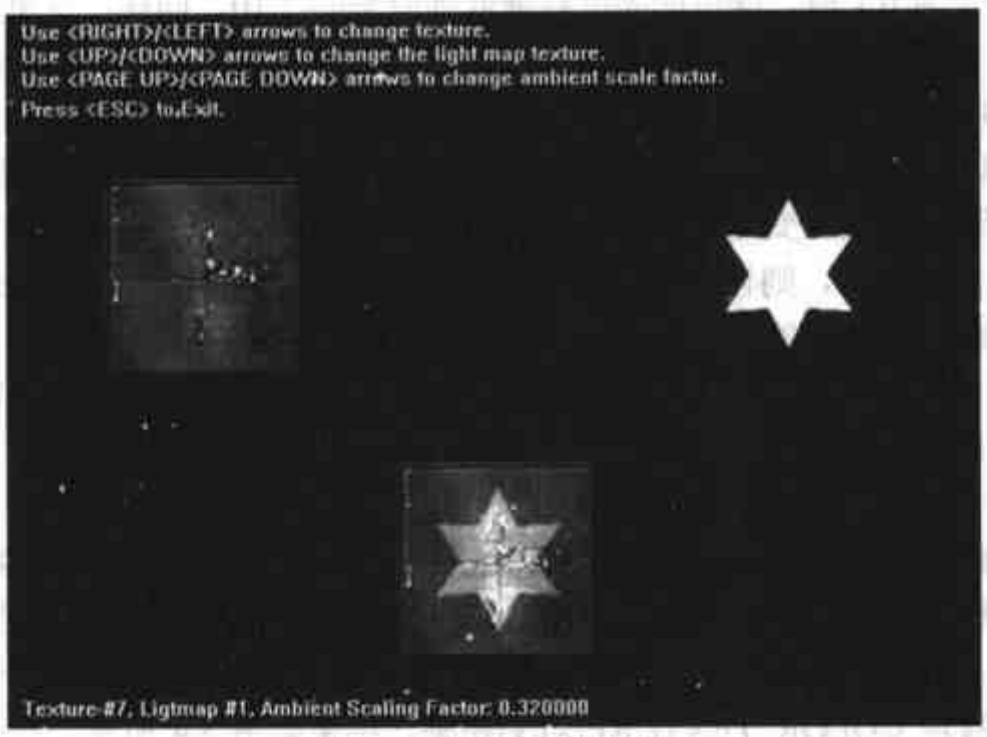


图 8.5　加法和乘法调制

鉴于本书将速度放在首位，我们已经看到了与光照相关的一些问题。即使是将光照纹理图（将在后面详细介绍）应用于纹理这样简单的情况，也需要执行大量的乘法和加法运算，还可能需要使用条件逻辑来检查溢出，因此必须减少计算量。我们必须使用尽可能好的数值模型，并尽可能减少为获得最终的像素值需要执行的转换。另外，如果使用 8 位颜色索引模式，工作将更加复杂，但在这种模式下可以使用查找表，因此很多处理的速度更快。

3. alpha 混合

最后要讨论的是 alpha 混合，它同时利用了相加和调制的概念。alpha 混合指的是将多个像素的颜色相加，得到目标像素的颜色，混合时将指定每个源像素的百分比。图 8.6 显示了两个源位图及其 alpha 混合的结果。alpha 混合用于模拟透明和其他效果（如阴影）。alpha 混合的计算非常简单，其工作原理如下：根据混合因子（通常被称为 alpha）将多个源像素混合起来，并将结果作为目标像素的颜色值。例如，下面是一个对两个源像素进行 alpha 混合的公式：

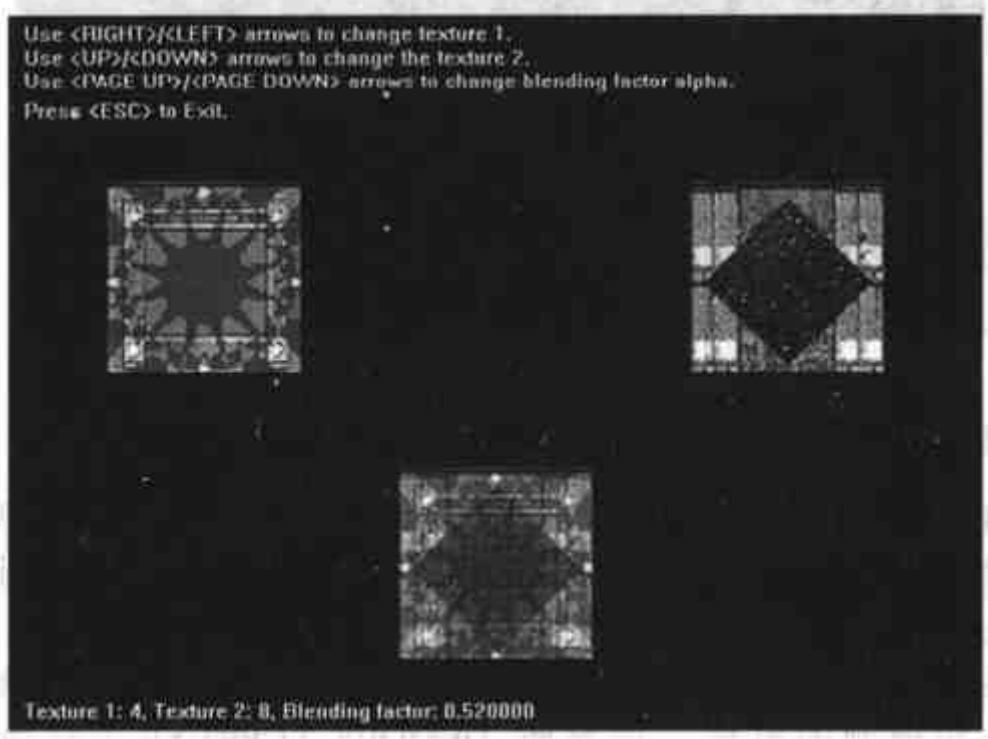


图 8.6　alpha 混合

$$pixel\_dest[x,y]_{rgb} = pixel\_source1[x,y]_{rgb} * alpha + pixel\_source2[x,y]_{rgb} * (1 - alpha)$$

如果 alpha = 1，则目标像素将是 pixel_source1[ ][ ]的拷贝；如果 alpha = 0.9，则目标像素的颜色值 90%来自 pixel_source1[ ][ ]，10%来自 pixel_source2[ ][ ]。不用说，必须对每个像素和每个颜色分量执行这种处理，但算法非常简单。DEMOII8_3.CPP|EXE 演示了 alpha 混合，读者可以使用左、右箭头键来修改 alpha 的值，以观察两个源纹理的混合结果。

至此，读者对如何使用颜色有一些了解，接下来讨论各种类型的光。

4. 环境光

环境光是来自四面八方的光，它没有确切的来源，而是各种光经过各种物体反射后的结果。环境光照项可以表示为环境光强度和 $I_{ambient}$ 和表面的环境反射颜色 $Cs_{ambient}$ 的乘积（很多文献都使用 k 来表示反射，但模拟光照时，将表面材质反射和光线颜色都视为颜色，将更好理解，因此这里使用 C）。同样，这一项可表示为 RGB 格式，对于单色光，也可以使用一个标量来表示其强度。场景中所有多边形都将受环境光的影响，而不管是否有光源。对于表面 S 上的某一点，反射的环境光强度为：

$$I_{totala} = Cs_{ambient} * I_{ambient}$$

注意：为简化数学计算，假设反射色为浮点数 RGB 格式（0...1, 0...1, 0...1），强度值为浮点数标量。另外，强度 I 的下标中有一个粗体字母，它表示要计算其最终强度的光类型，上面的 a 表示环境光。

5. 散射光

散射光是被物体散射出去的光。散射光照与观察者的位置无关，沿所有方向均匀地散射。在图 8.7 中，一束光照射到表面 S 上，后者沿所有方向将光束散射出去，但散射光强度取决于表面和光源之间的相对角度。最后的散射光强度与面法线和光源向量之间的夹角余弦呈正比。

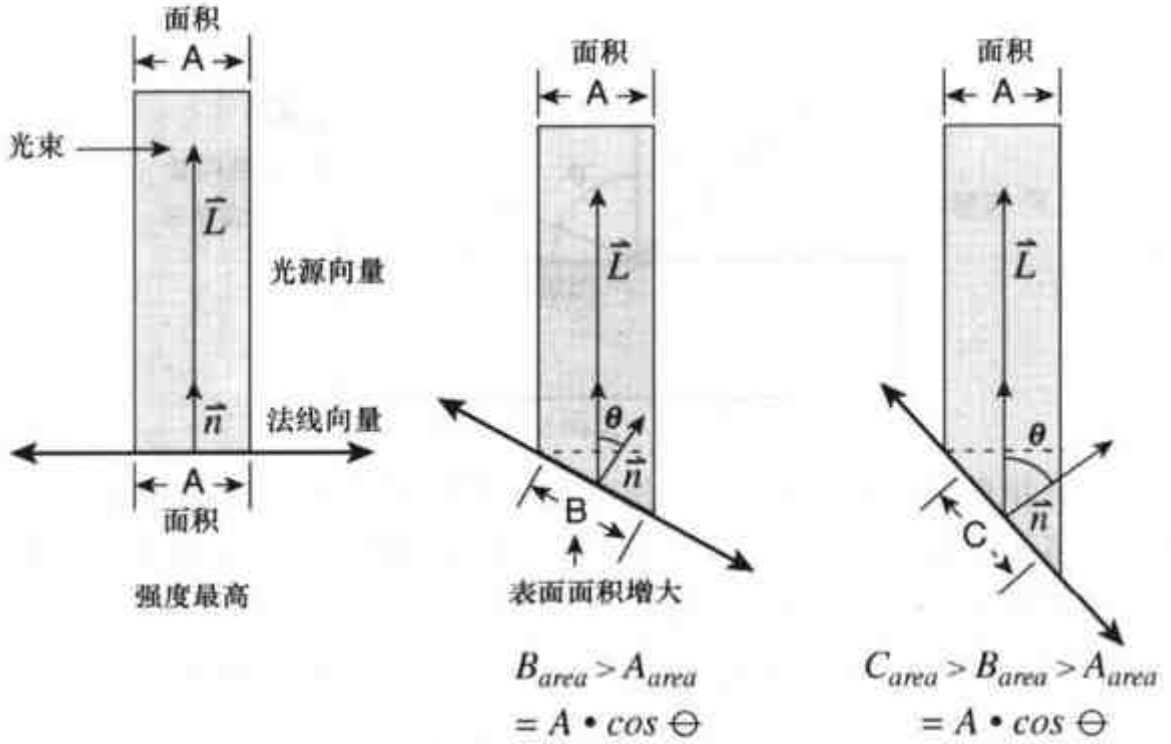


图 8.7　漫反射几何学

这是因为随着表面越来越接近于与光源向量平行，表面面积越来越大，因此每单位面积反射的能量越来越小。相反，随着表面越来越接近于与入射光垂直，表面面积越来越小，每单位面积反射的能量越来越大。最后，当法线向量 *n* 与光源向量垂直时，散射光强度最大。

漫反射也被称为 Lambertian 反射，它基于这样一种概念，即表面是由以不同方式同光线交互的材质组成。其中一种方式是，沿所有方向反射光线中的某些分量，这是由于表面的各个微小部分的朝向是随机的。实际上，这种物理现象要复杂得多。光线的散射或多或少地与频率相关，散射角可能随频率而异。然而，在我们的模型中，将假设每种表面材质都有特定的漫反射颜色 $Rs_{diffuse}$，它决定了该表面材质对散射光的反射程度。$Rs_{diffuse}$ 是一种 RGB 颜色，而不是标量，这意味着表面材质很可能只散射某种频率的光。例如，$Rs_{diffuse}$ 为(0，0，1.0)时，将只散射蓝色光；表面可能等量地散射所有颜色的光，且没有其他类型的光散射（如镜面反射），此时 $Rs_{diffuse}$ 为(1.0，1.0，1.0)。使用漫反射颜色 $Rs_{diffuse}$ 和光源的散射光强度 $I_{diffuse}$ 表示时，

单光源的散射光照方程如下：

$$I_{totald} = Rs_{diffuse} * I_{diffuse} * (n \cdot l)$$

加入散射项后，单光源的光照模型如下：

$$I_{totalad} = Rs_{ambient} * I_{ambient} + [Rs_{diffuse} * I_{diffuse} * (n \cdot l)]$$

在有多个光源的情况下，只需将每个光源的散射项累积起来，而环境光项保持不变。

$$I_{totalad} = Rs_{ambient} * I_{ambient} + Rs_{diffuse} * \sum_{i=1}^{n} [I(i)_{diffuse} * (n_i \cdot l_i)]$$

注意：读者可能注意到了，一些表示方法是冗余的。例如，没必要同时使用强度和反射率（尤其是对环境光项而言），可以将计算它们的乘积，并用单个常量表示结果。然而，这里旨在指出各个独立的参数，而不是要使用最优化的表示方法。

6. 镜面反射光

镜面反射在很大程度上是由于物体表面上大量微型面的朝向相同导致的。镜面反射光照模型旨在模拟这种效果，这与光源、面法线和观察位置相关（在散射光照中，观察位置无关紧要）。

图 8.8 说明了镜面反射光照，其中显示了表面 S、面法线 **n**、观察向量 **v** 和反射向量 **r**（它总是与光照向量 l 相对于面法线对称）。作为一个试验，读者可尝试这样做：将一个光亮物体放在眼前，并缓慢地旋转它或移动脑袋，直到在物体中看到房间中的某个光源——这就是镜面反射。

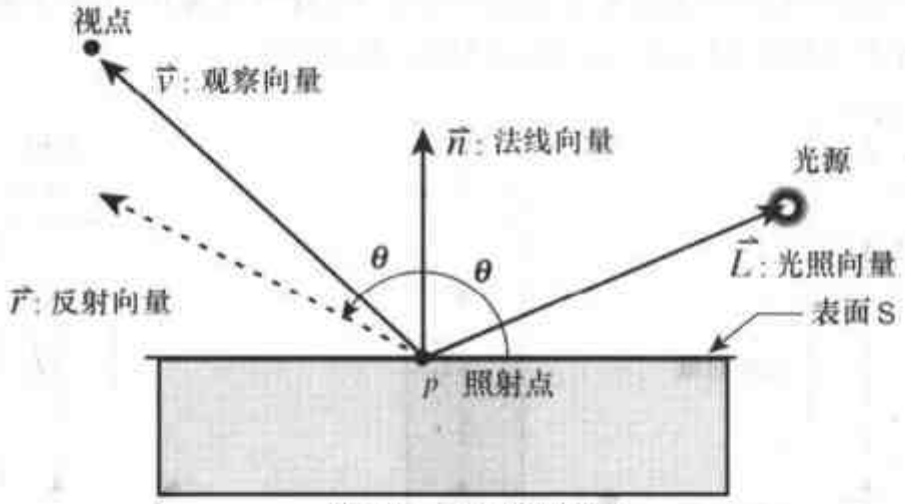


图 8.8 镜面反射光照

当光源的反射向量 r 和观察向量 v 重叠时，您将看到光源的镜像，因为物体表面就像一面镜子。物体的镜面反射系数越大，镜面反射区域越小。当反射向量 r 和观察向量 v 的夹角接近于 0 时，反射光强度将急剧增大；而随着夹角逐渐增大，镜面反射光强度将急剧降低。

提示：这种模型通常被称为 Phong 光照模型；另外通过插值计算法线以进行着色时被称为 Phong 着色。这是两个不同的概念，有时会让人混淆，因为它们是相关的。

为模拟镜面反射光照，需要考虑观察向量和反射向量以及一个定义物体表面光洁度的常量（镜面反射率，specularity）。常识告诉我们，镜面反射光强度随观察向量和反射向量之间的夹角减小而增大，同时如果面法线和光照向量的夹角大于 90 度，光源将照射不到表面。

因此，需要知道反射系数，我们称之为 $Rs_{specular}$；还需要知道光照强度 $I_{specular}$。最后，有时候还需要知道镜面反射指数（specular exponent），它决定了镜面反射区域的大小，我们将这个变量称为 sp。这样，单光源的镜面反射光照方程如下：

$$I_{totals} = Rs_{specular} * I_{specular} * MAX(r \cdot v, 0)^{sp}$$

注意，这里使用函数 MAX() 只为避免值小于 0。

提示：这里使用的是点积表示，也可以使用点积的定义：$u \cdot v = |u|*|v|*\cos\alpha$。这样上述方程将变为 $Rs_{specular} * I_{specular} * \cos^{sp}\alpha$。当然，这里假设 r 和 v 都是单位向量。

正如前面指出的，光照向量和面法线的夹角必须小于 90 度，即它们的点积必须大于 0。因此，如果您愿意，可以在上述方程中使用布尔运算符[(*n.l*)>0 ? 1: 0]来避免对于 S 面上那些光源照射不到的点计算镜面反射光照：

$$I_{totals} = Rs_{specular} * I_{specular} * MAX(r \cdot v, 0)^{sp} * [(n.l)>0 \ ? \ 1: \ 0]$$

接下来看看多光源的情况。和前面一样，只需将每个光源的贡献相加即可：

$$I_{totals} = Rs_{specular} * \sum_{i=1}^{n}[I(i)_{specular} * MAX(r_i \cdot v_i, 0)^{sp} * [(n_i \cdot l_i)>0?1:0]]$$

将环境光照项、散射项和镜面反射项相加后，光照模型如下：

$$I_{totalads} = I_{totala} + I_{totald} + I_{totals} =$$

$$Rs_{ambient} * I_{ambient} + Rs_{diffuse} * \sum_{i=1}^{n}[I(i)_{diffuse} *(n_i \cdot l_i)] +$$

$$Rs_{specular} * \sum_{i-1}^{n}[I(i)_{specular} * MAX(r_i \cdot v_i, 0)^{sp} * [(n_i \cdot l_i)>0?1:0]]$$

7. 发射光

发射光是所有光照模型中最容易实现的。设置表面的反射率时，指定了它发出的光强度，即它本身是发光的。如果这种光是“真正的”光，需要将这个表面视为光源，并在执行光照计算时，通过散射光照方程和镜面反射光照方程中包含它和其他光源。但如果不考虑发射光对其他物体的影响，则只需使用表面的发射率来计算该表面的光照即可。

例如，假设游戏世界中有一个灯塔，灯塔可能是由环境光反射系数和散射光反射系数都很小的材质组成，这样当光源在周围移动时，将只能看到灯塔的大概形状；但打开灯塔上的灯后，灯塔将被照亮。然而，灯塔并非光源，并不向外发射光；但看起来像是光源。图 8.9 说明了这一点：图中有一个灯塔，灯塔的旁边有一个物体，虽然灯塔是亮的，但并没有光照射到物体上。实现反射光非常简单：

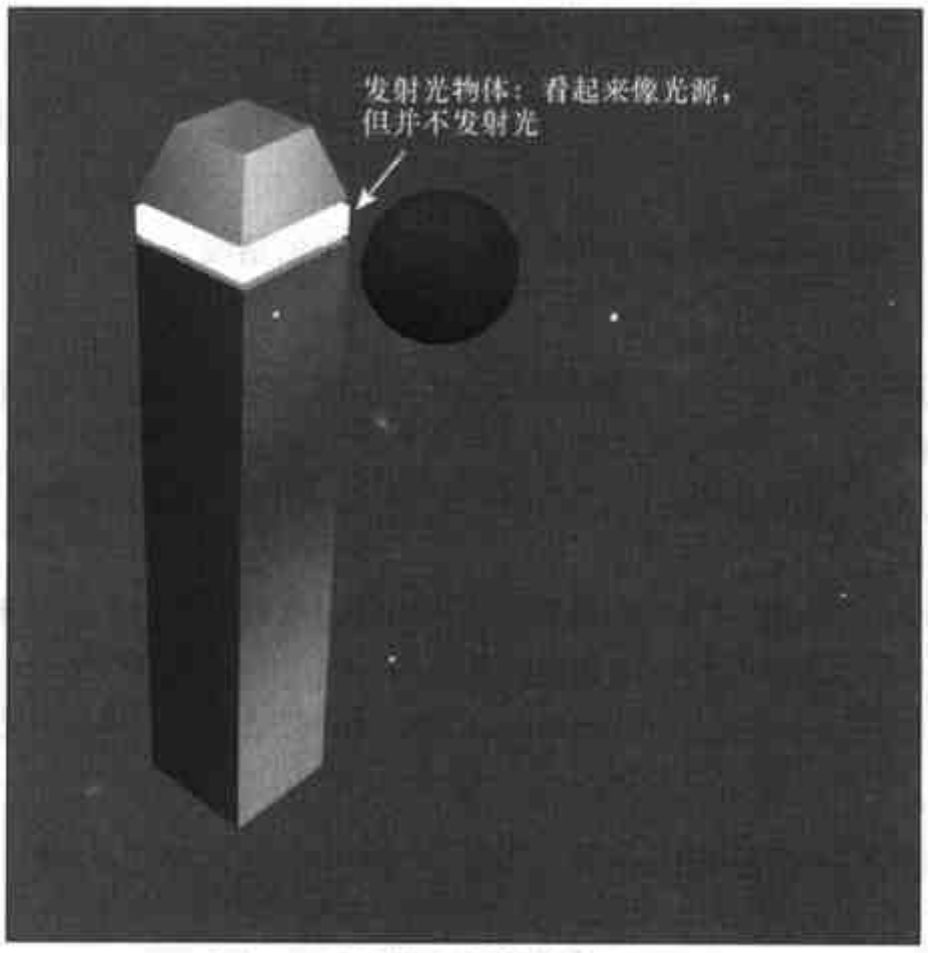


图 8.9　不发射光线的发射/固定发照

$$I_{totale} = Rs_{emission}$$

即发射光对像素颜色的贡献为 $Rs_{emission}$。这样，完整的光照方程如下：

$$I_{totalaeds} = Rs_{ambient} * I_{ambient} + Rs_{emission} + Rs_{diffuse} * \sum_{i=1}^{n}[I(i)_{diffuse} * (n_i \cdot l_i)] + Rs_{specular} * \sum_{i=1}^{n}[I(i)_{specular} * MAX(r_i \cdot v_i, 0)^{sp} * [(n_i \cdot l_i)>0?1:0]]$$

最后需要指出的一点是，光照强度 $I_{ambient}$、$I_{diffuse}$ 和 $I_{specular}$ 可能相同，也可能不同。有些光源可能只影响光照模型的散射部分，而有些光源只影响镜面反射光部分。然而，在很多情况下，光源的 $I_{diffuse}$ 和 $I_{specular}$ 相同；假设第 i 个光源的这些光照强度为 I_i，则上述方程可简化为：

$$I_{totalaeds} = Rs_{ambient} * I_{ambient} + Rs_{emission} + Rs_{diffuse} * \sum_{i=1}^{n}[I(i)_{diffuse} * (n_i \cdot l_i)] + Rs_{specular} * \sum_{i=1}^{n}[I(i) * MAX(r_i \cdot v_i, 0)^{sp} * [(n_i \cdot l_i)>0?1:0]]$$

必须对每个像素执行光照计算，并考虑场景中的每个光源。这涉及到的工作量非常大，而我们还没有讨论过光源。前面讨论的是光照，而不是光源。接下来需要讨论光源本身以及如何计算每个光源的光照强度。这又将是一场恶战。

8.1.2 光源类型

即使使用极度简化的光照模型，在多边形网格不多的情况下，结果也是合理的，看起来引人入胜。而另一方面，光照是基于像素的操作，即使使用粗糙的光照模型，仍需要对每个像素执行大量的运算，才能得到正确的光照效果。使用软件来实现这种功能时，涉及到很多窍门。

接下来要讨论的是光源。读者可能认为光源就像办公室和起居室中的梨状的东西，但数学模型截然不同。在大多数情况下，计算机模型只使用三种光源：

- 定向光源；
- 点光源；
- 聚光灯。

令人欣慰的是，上述每种光源都粗略地模拟了现实生活中的情况。

1. 定向光源

定向光源是没有确切位置的光源，它离场景非常远（实际上是无穷远），其光线平行地照射到表面上，如图 8.10 所示。这种光源也被称为无穷远光源。定向光源还具有这样的特征：其光照强度不会随距离而衰减，这是因为光线已经传播了无穷远的距离。然而，定向光源仍然有强度和颜色。要定义定向光源，只需指定其初始颜色和强度即可。

读者需要注意表示方法。前面使用了表面材质的环境光反射系数、散射光反射系数和镜面反射光属性，它们是表面材质本身的反射系数，而现在讨论的是光源本身的颜色，不要将它们混为一谈。为此，我们使用 Cl_{xxx} 来表示光源颜色，其中 xxx 是光源类型；同时用 $I0_{xxx}$ 表示光源的初始强度。因此，定向光源的强度为：

$$I(d)_{dir} = I0_{dir} * Cl_{dir}$$

定向光源的光照强度不随距离而变化，因此上述方程中不包含距离。

2. 点光源

点光源是使用 3D 空间中的一个点来模拟的，如图 8.11 所示。点光源实际上被模拟为现实生活中的点光源，因此光照强度随光源离表面 S 的距离衰减，而衰减是使用三个因子来模拟的：常量衰减因子（k_c）、线性衰减因子（k_l）和二次衰减因子（k_q）。用 d 表示位于点 p 处的点光源和表面 S 上的点 s 之间的距离，则光照强度和距离之间的关系如下：

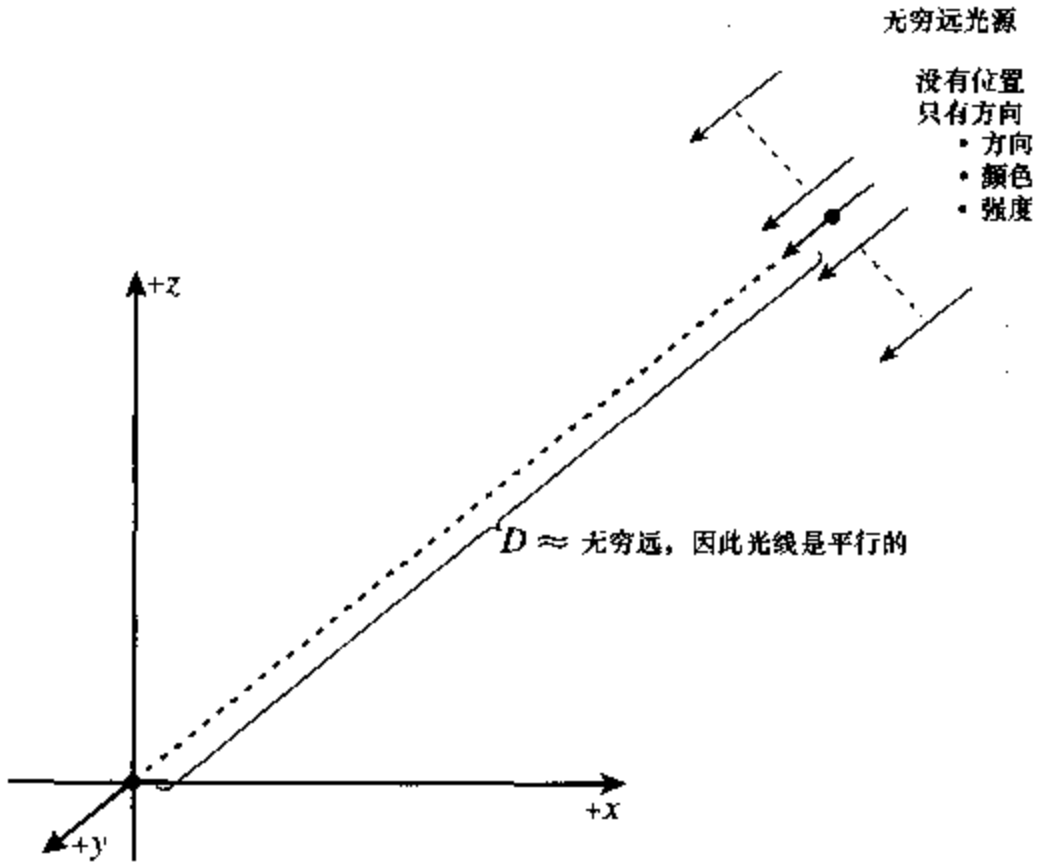


图 8.10　无穷远光源的特征

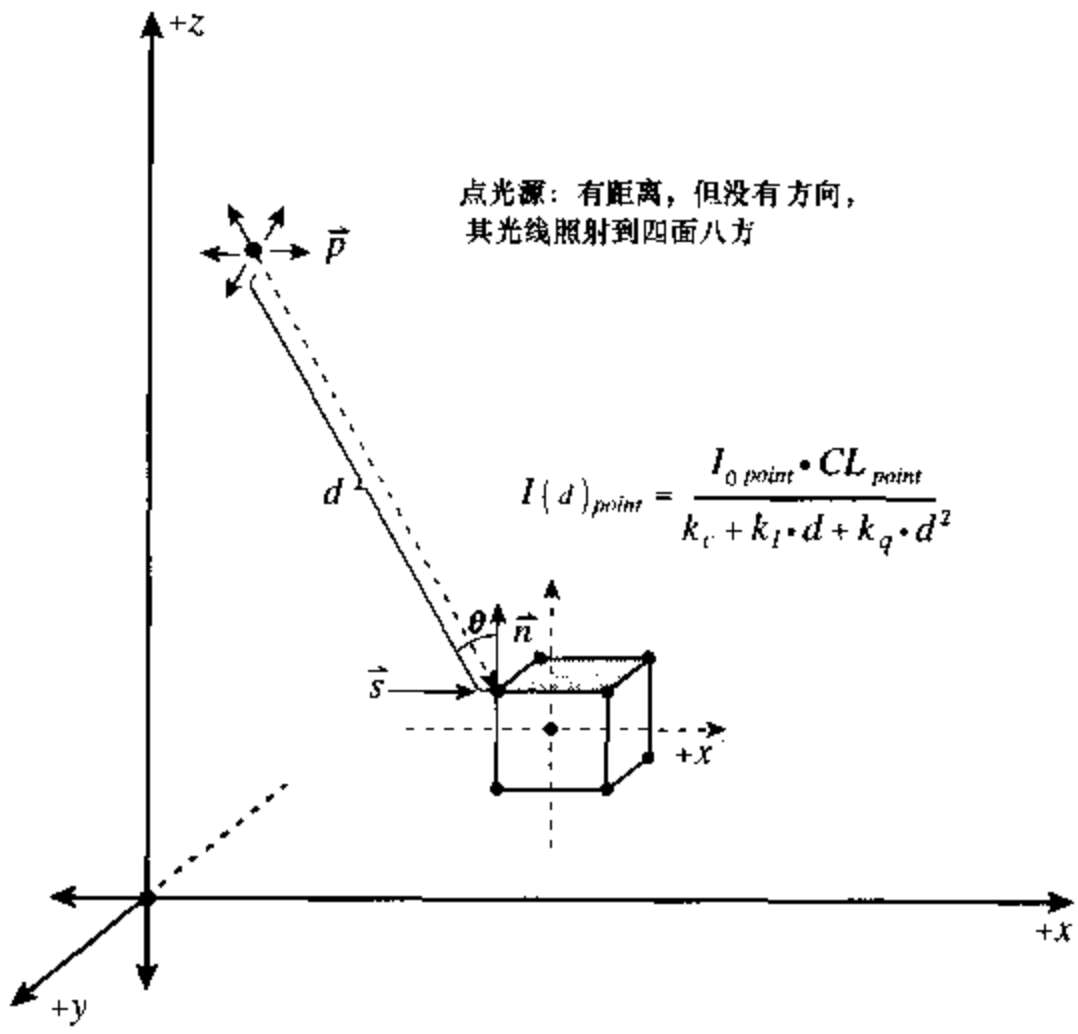


图 8.11　点光源的特征

$$\mathrm{I(d)_{point}} = \frac{I0_{\mathrm{point}} * Cl_{\mathrm{point}}}{k_c + k_l * d + k_q * d^2}$$

其中 d = |p − s|

为让读者对这个函数有感性认识，下面通过一些具体的数字来计算其结果。需要指出的是，所有的光照方程都是基于 RGB 的，也就是说，每个方程实际上表示三个方程：它们分别对应于红色、绿色和蓝色分量。然而，由于这三个方程相同，因此只写出其中的一个，但读者应该明白它其实代表三个方程。因此，可以通过计算一个分量的结果（即假设系统为单色的）来了解 RGB 的变化曲线（即不失普遍性）。首先，

假设 $I0_{point} = 1.0$，$Cl_{point} = 1.0$，这样它们的乘积为 1.0，从而确保结果是归一化的。接下来使用三组不同的衰减因子：

- 第 1 组：$k_c = 1$，$k_l = 0$，$k_q = 0$；
- 第 2 组：$k_c = 0$，$k_l = 1$，$k_q = 0$；
- 第 3 组：$k_c = 0$，$k_l = 0$，$k_q = 1$。

图 8.12 显示了这三组衰减因子对应的衰减曲线。正如读者看到的，在线性衰减因子为 1 和二次衰减因子为 1 这两种情况下，光照强度的衰减速度都非常快。事实上，当距离为 10 且 $k_q = 1$ 时，光照强度只有原来的 1%左右，几乎是没有光照。因此，请务必谨慎设置衰减因子。作者建议只使用线性衰减因子，并将其设置为 0.001～0.0005。

距离	1/d	$1/d^2$	$\frac{1}{1+1/d+1/d^2}$
d = 0.500:	1/d = 2.000,	1/d^2 = 4.000,	1/(1+d+d^2) = 0.571
d = 1.000:	1/d = 1.000,	1/d^2 = 1.000,	1/(1+d+d^2) = 0.333
d = 1.500:	1/d = 0.667,	1/d^2 = 0.444,	1/(1+d+d^2) = 0.211
d = 2.000:	1/d = 0.500,	1/d^2 = 0.250,	1/(1+d+d^2) = 0.143
d = 2.500:	1/d = 0.400,	1/d^2 = 0.160,	1/(1+d+d^2) = 0.103
d = 3.000:	1/d = 0.333,	1/d^2 = 0.111,	1/(1+d+d^2) = 0.077
d = 3.500:	1/d = 0.286,	1/d^2 = 0.082,	1/(1+d+d^2) = 0.060
d = 4.000:	1/d = 0.250,	1/d^2 = 0.063,	1/(1+d+d^2) = 0.048
d = 4.500:	1/d = 0.222,	1/d^2 = 0.049,	1/(1+d+d^2) = 0.039
d = 5.000:	1/d = 0.200,	1/d^2 = 0.040,	1/(1+d+d^2) = 0.032

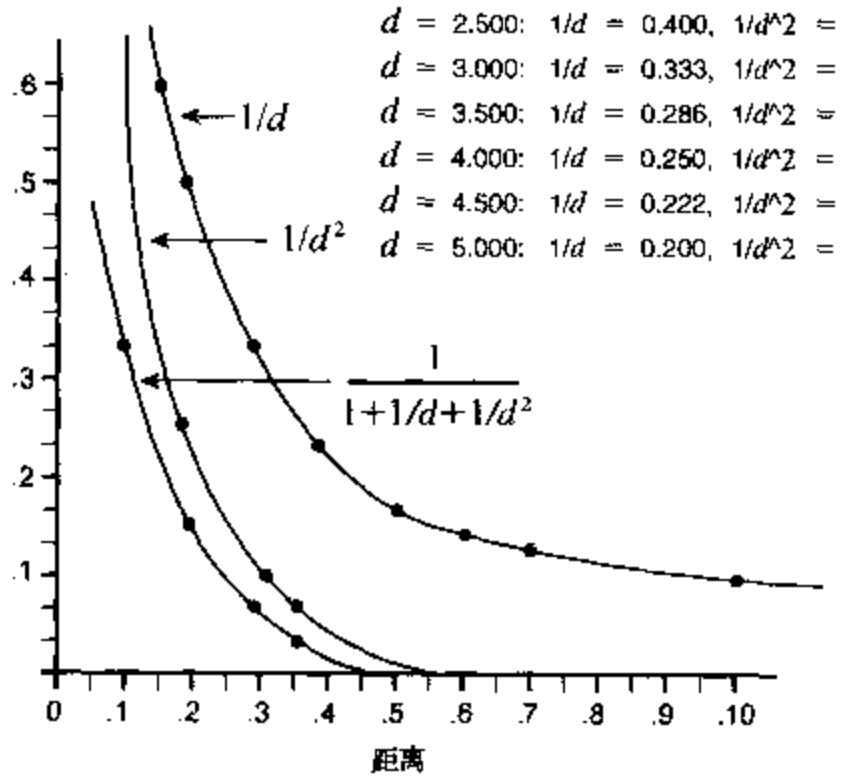


图 8.12　衰减曲线

当然，当所有衰减因子都不为零时，强度衰减速度将更快。然而，在实际应用中，作者发现不包含二次项的更简单的光源模型也很好使：

$$I(d)_{point} = \frac{I0_{point} * Cl_{point}}{k_c + k_l * d}$$

其中 d = |p – s|

3. 聚光灯

这里要讨论的最后一种光源是聚光灯，如图 8.13 所示。聚光灯的计算开销非常高（即使使用硬件），要在软件中模拟它，必须使用一些窍门并对其进行简化；不过现在暂时不去管它，先来看看其数学原理。图 8.14 是描述了标准的聚光灯模拟方式，其中聚光灯位于点 p 除，方向向量为 l，光线照射到表面 S 上的点 s；另外还有一个表示聚光灯照射范围的圆锥。这个锥体由两个区域组成：内部区域和外部区域，前者是由角度α定义的，通常被称为本影（umbra），后者（柔和阴影部分）是由角度 ϕ 定义的，被称为半影（penumbra）。

在本影内，光照强度是恒定的，但过渡到半影中后急剧衰减。为模拟这种衰减方式，大多数引擎都使用下述算法：

情形 1　如果光源方向与光源到点 s 的向量之间的夹角大于外圆锥（半影）的角度，则不对该点执行光照计算；

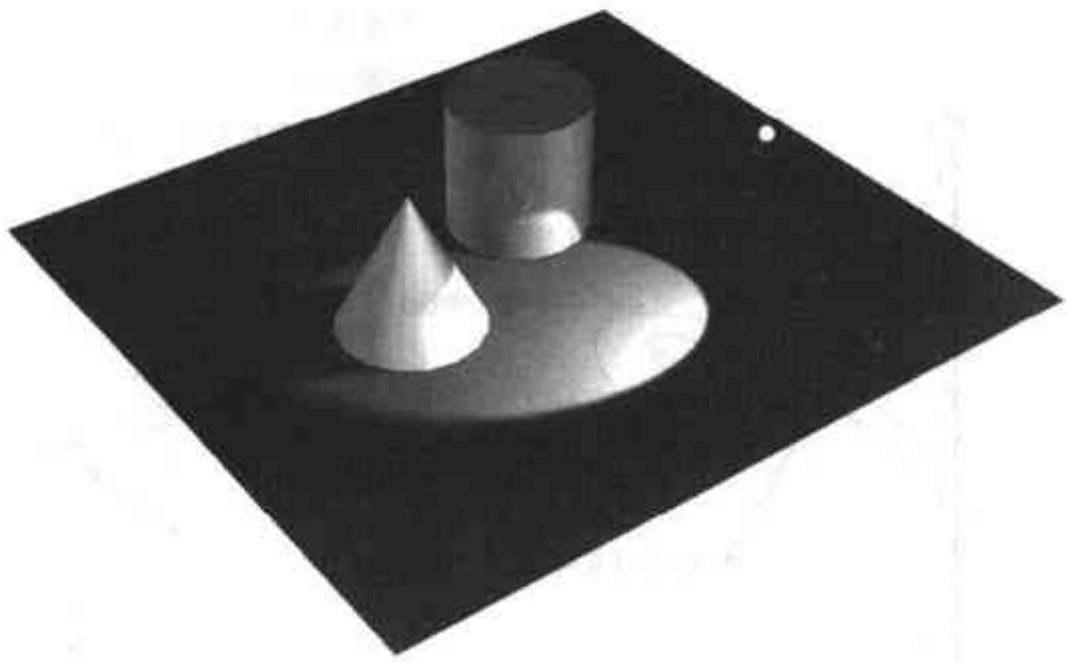


图 8.13　照射物体的聚光灯

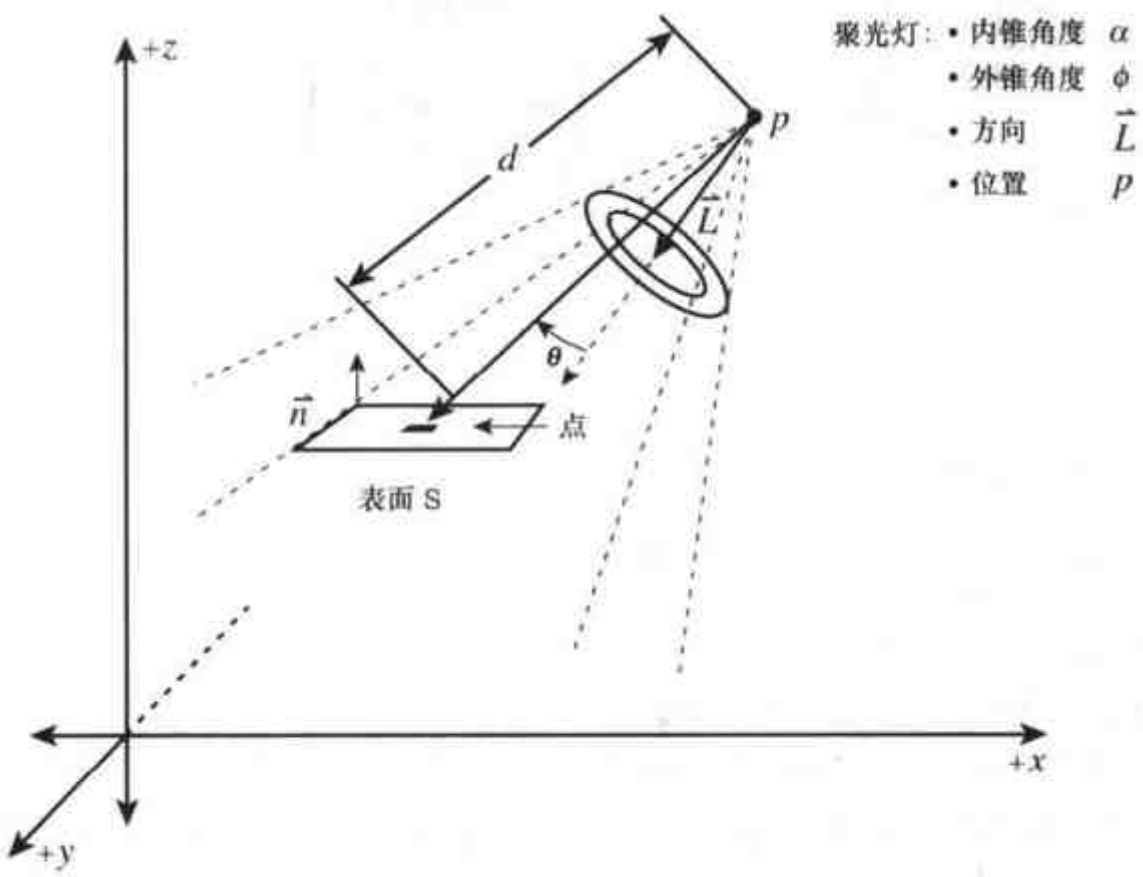


图 8.14　聚光灯光照计算

情形 2　如果光源方向与光源到点 s 的向量之间的夹角小于内圆锥（本影）的角度，则使用 100%的聚光灯强度，但根据距离对其进行衰减；

情形 3　如果光源方向与光源到点 s 的向量之间的夹角大于内圆锥（本影）的角度，但小于外圆锥（半影）的角度，则根据距离和减弱（falloff）公式对强度进行衰减。

这里考虑了聚光灯的方方面面，但问题是，有必要这样做吗？是否可以使用一个更简单的模型获得相同的效果？这是可能的，稍后将介绍一个这样的模型。但现在还是来探讨这个复杂的模型，Direct3D 使用的就是这种模型。

要定义聚光灯，需要指定位置（p）、方向向量（l）和两个角度（α和ϕ，它们分别定义了内锥和外锥）。另外，为简化角度比较工作，我们使用两个半角：$\alpha^* = \alpha/2$ 和 $\phi^* = \phi/2$。当然，还需要光源颜色 $Cl_{spotlight}$ 和强度 $I0_{spotlight}$。

图 8.15 描述了聚光灯模型中各个要素之间的几何关系。因此，如果角度 θ 大于 ϕ^*，聚光灯根本就照射不到相应的点；如果 θ 小于 ϕ^*，相应的点位于照射范围内——但内锥中，无需考虑减弱，而在外锥中需要。

还需要指定一个控制聚光灯总体光照强度的指数因子（power factor），用 pf 表示。最后，由于聚光灯属于点光源，需要考虑光照强度随距离衰减的问题，因此需要指定衰减因子 k_c、k_l 和 k_q。这样计算聚光灯光照强度的公式如下：

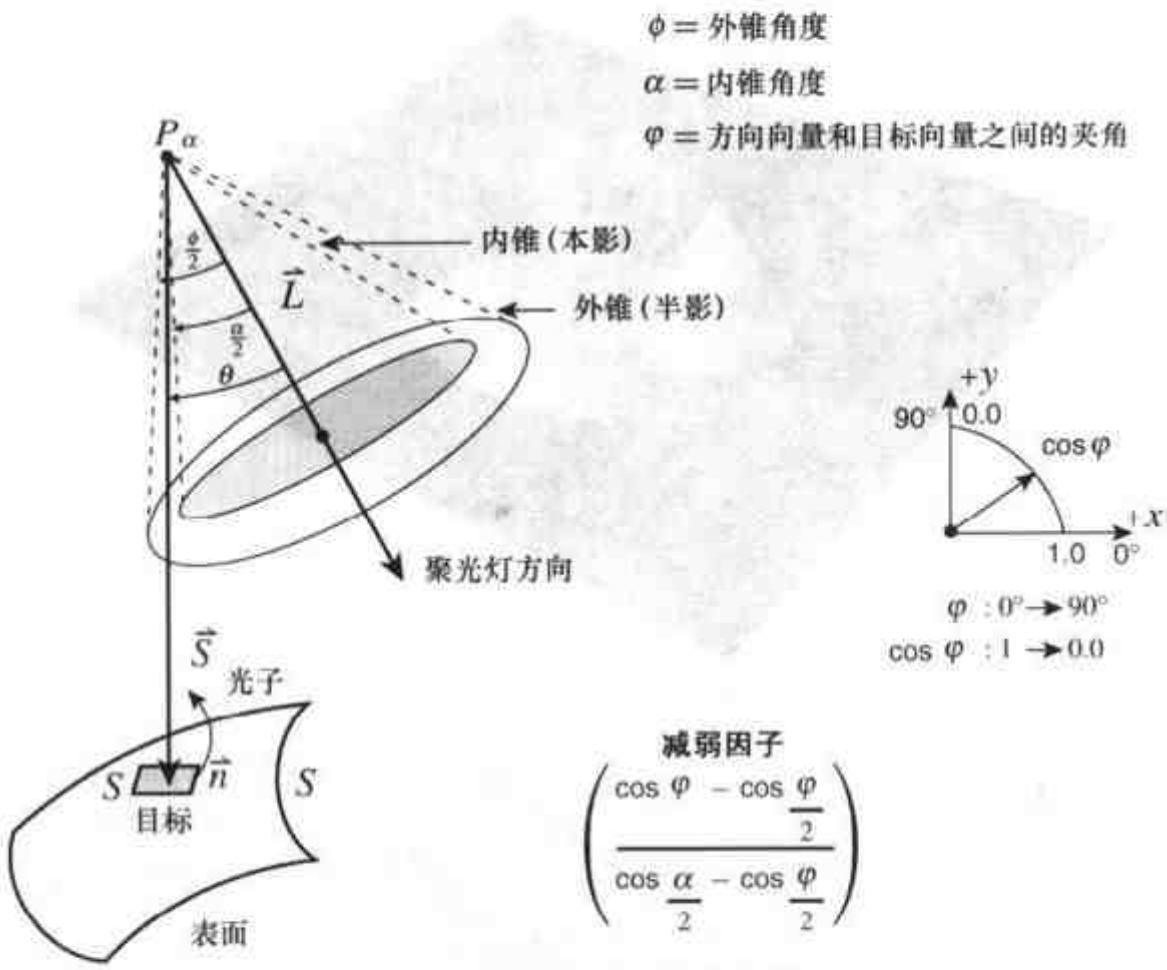


图 8.15　对聚光灯的详细数学分析

情形 1　$\theta > \phi^*$（位于半影外，根本照射到）：

$I(d)_{spotlight} = 0$

情形 2　$\theta <$（位于本影内）：

$$I(d)_{spotlight} = \frac{I0_{spotlight} * Cl_{spotlight}}{k_c + k_l * d + k_q * d^2}$$，其中 d = |p − s|

情形 3　$\alpha^* < \theta < \phi^*$（位于半影内）：

$$I(d)_{spotlight} = \frac{I0_{spotlight} * Cl_{spotlight}}{k_c + k_l * d + k_q * d^2} * \frac{(\cos\theta - \cos\phi^*)^{pf}}{(\cos\alpha^* - \cos\phi^*)}$$，其中 $d = |p - s|$，pf 为指数因子。

读者可能会说，“这有点不太好懂，我能理解点积，但不知道最右边的那项有何含义”。确实如此。下面来看这一项是否合理。假设本影角度α为 40 度，半影角度β为 60 度，而角度θ的范围为 $20 < \theta < 40$。将它们代入到上述光照公式中最右边的一项中，结果如下：

$$I = \frac{(\cos\theta - \cos 30)^{1.0}}{(\cos 20 - \cos 30)}$$

出于简化的目的，这里假设指数因子为 1.0。角度接近零时，其余弦接近 1.0；而角度接近 90 度时，其余弦接近零。我们希望角度θ 接近内锥半角时，上述比值接近 1.0，而θ接近外锥半角时，上述比值接近 0.0。来看看上述比值是否是这样的。如果θ等于内锥半角（20 度），则结果为：

$$I = \frac{(\cos 20 - \cos 30)^{1.0}}{(\cos 20 - \cos 30)} = 1.0$$

这正是我们希望的。如果θ等于外锥半角，比值应该为 0.0：

$$I = \frac{(\cos 30 - \cos 30)^{1.0}}{(\cos 20 - \cos 30)} = 0.0$$

确实如此。那么，减弱曲线的中间是什么样的呢？图 8.16 是 pf 为 0.2、1.0 和 5.0 对应的减弱曲线，其中内锥角为 30 度，外锥角为 40 度，θ的变化范围为 15～20 度。在大多数情况下，将指数因子设置为 1.0

是可行的。表 8.1 列出了绘制图 8.16 时使用的数据。

表 8.1　　聚光灯减弱曲线数据

指数因子	θ	减弱因子	指数因子	θ	减弱因子
0.200000			1.000000		
	15	1.000000		15	1.000000
	16	0.961604		16	0.822205
	17	0.912672		17	0.633248
	18	0.845931		18	0.433187
	19	0.740122		19	0.222083
	20	0		20	0.000000
5.000000			5.000000		
	15	1.000000		18	0.015254
	16	0.375752		19	0.000540
	17	0.101829		20	0.000000

现在的问题是，有必要使用这样复杂的模型吗？如果是渲染离线（offline）3D 环境，完全有必要；如果是实时地渲染 3D 游戏，则可能没有必要。在实时游戏中，非常粗略地模拟聚光灯的模型就能满足需要。事实上，只需这样做即可：不将锥体划分成强度不减弱的本影和减弱的半影，而是将整个锥体内的减弱因子都设置为 $\cos\theta$，同时考虑指数因子。这样，简化后的模型如下：

$$I(d)_{spotlight} = \frac{I0_{spotlight} * Cl_{spotlight} * MAX(\cos\theta,0)^{pf}}{k_c + k_l * d + k_q * d^2}$$

其中 $d = |p - s|$，pf 为指数因子。

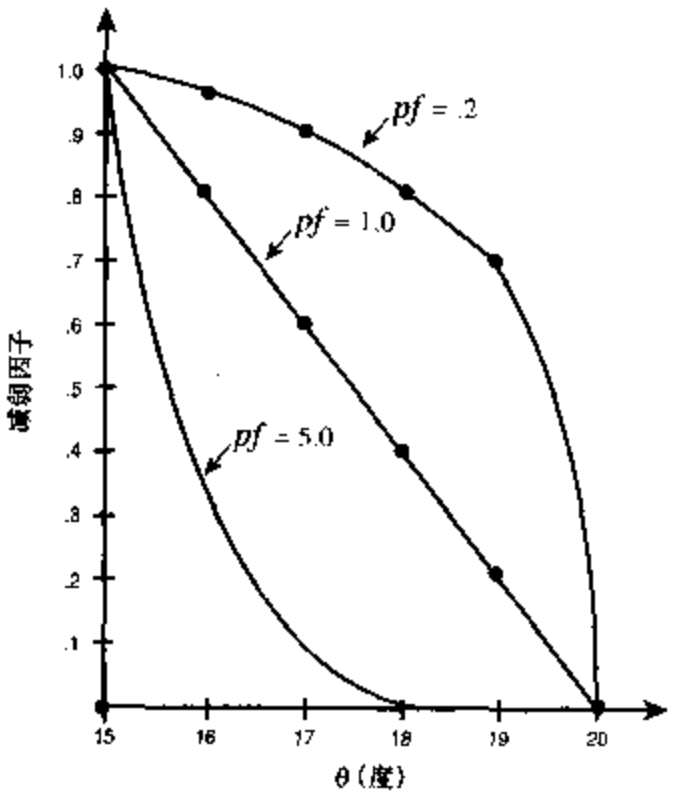


图 8.16　聚光灯强度的减弱曲线

4. 光照小结

正如读者看到的，光照涉及的内容很多。读者需要牢记的是，光照旨在确定多边形中每个像素的颜色。为此，我们推导了一些模型，以便能够使用诸如对环境光、散射光和镜面反射光的反射系数等属性来定义虚拟材质。另外，还需要有虚拟光源，用于发射同虚拟材质交互的光。光源可能简单如无处不在的环境光光源，也可能复杂如聚光灯。

现在，我们的引擎根本不支持材质和光源，因此必须添加这样的支持。我们会这样做的，不过重要的事先办，先在线框引擎中加入绘制实心多边形的功能，因为使用光照时，需要将多边形绘制为实心的。

注意：本章涉及的软件和结构等都可在 T3DLIB6.CPP|H 中找到。

8.2　三角形的光照计算和光栅化

在《Windows 游戏编程大师技巧》的第 8 章，作者介绍了如何使用纯色光栅化三角形和多边形，这里不想再重复，因为在本书的第 9 章介绍纹理映射时，将更深入地探讨这方面的内容。这里只简要地回顾这方面的内容，供没有阅读过《Windows 游戏编程大师技巧》一书的读者参考。首先，这个 3D 引擎只处理三角形，因此无需考虑四边形及更复杂的多边形。图 8.17 是一个要对其进行光栅化的三角形。

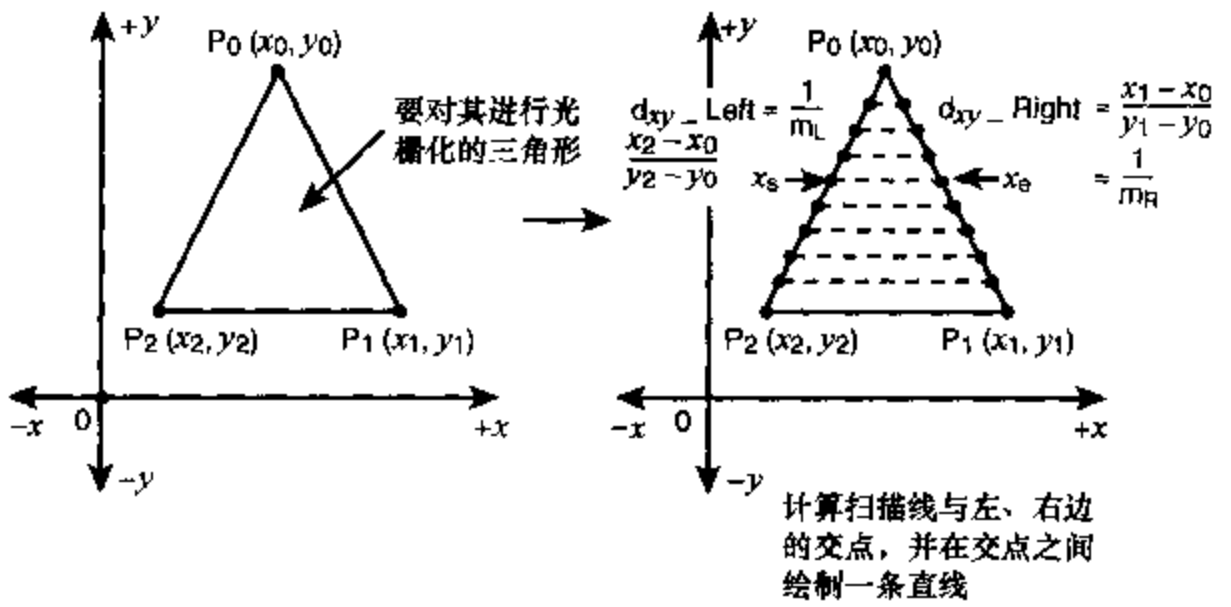


图 8.17　三角形的光栅化

绘制三角形与绘制直线非常相似：必须确定三角形的左边（left edge）和右边（right edge）上要绘制的点，并在这两点之间绘制一条直线；然后再绘制下一条直线，依次类推，如图 8.17 所示。正如读者看到的，计算出每条边的斜率后，只需移到下一跳扫描线（即将 y 值加 1——译者注），并根据斜率（准确地说是斜率的倒数）调整端点的 x 坐标（xs 和 xe），然后绘制一条直线即可。

下面的算法假定要光栅化的是平底三角形：

第 1 步　计算左边和右边的 dx/dy，这相当于斜率的倒数。之所以计算它们是因为将使用一种垂直移动（vertically oriented）的方法，需要知道 y 值变化一个单位时，*x* 值将变化多少——*dx/dy* 或 1/*M*（其中 *M* 为斜率）。对于左边和右边的这种值，分别用 dxy_left 和 dxy_right 表示。

第 2 步　从最上面的顶点(x0，y0)开始，将 *xs* 和 *xe* 设置为 *x*0，*y* 设置为 *y*0。

第 3 步　将 *xs* 加 dxy_left，将 *xe* 加 dxy_right，将 *y* 加 1，得到要绘制的直线端点的坐标。

第 4 步　在(*xs*，*y*)和(*xe*，*y*)之间绘制一条直线。

第 5 步　回到第 3 步，直到达到并绘制三角形的底边为止。

当然，为正确设置初始条件和边界条件，需要花些时间来考虑。仅此而已，是不是很简单？

下面使用浮点数来实现平底三角形的光栅化算法，因为当前的计算机速度已经足够快。算法如下：

```
// 计算斜率倒数
float dxy_left  = (x2-x0)/(y2-y0);
float dxy_right = (x1-x0)/(y1-y0);

// 设置扫描线起点和终点的 x 坐标
float xs = x0;
float xe = x0;

// 绘制每条扫描线
for (int y=y0; y <= y1; y++)
    {
    // 使用颜色 c 绘制一条直线，该直线的 y 坐标为 y，起点和终点的 x 坐标分别为 xs 和 xe
    Draw_Line((int)xs, (int)xe, y, c);
    // 移到下一条扫描线
    xs+=dxy_left;
    xe+=dxy_right;
    } // end for y
```

下面讨论这种算法的一些细节和遗漏的地方。首先，该算法对端点坐标进行了截尾，这可能是糟糕的，因为这样做丢弃了信息。一种更好的方法是，将端点坐标转换为整数之前，将其加上 0.5。另一个问题与初始条件有关，在第一次循环中，绘制了一条只有一个像素的直线。这样做是可行的，但绝对需要优化。

对于平顶三角形，如何绘制呢？只需将最上面的两个顶点标记为 **p0** 和 **p1**，将最下面的顶点标记为 **p2**，然后对前述算法中的初始条件进行细微的修改，以正确地计算扫描线与左边和右边的交点即可。修改后的算法如下：

```
// 计算斜率倒数
float dxy_left  = (x2-x0)/(y2-y0);
```

```
float dxy_right = (x2-x1)/(y2-y1);

// 设置扫描线起点和终点的 x 坐标
float xs = x0;
float xe = x1;

// 绘制每条扫描线
for (int y=y0; y <= y2; y++)
    {
// 使用颜色 c 绘制一条直线，该直线的 y 坐标为 y，起点和终点的 x 坐标分别为 xs 和 xe
Draw_Line((int)(xs+0.5), (int)(xe+0.5), y, c);
// 移到下一条扫描线
xs+=dxy_left;
xe+=dxy_right;

} // end for y
```

至此，有了能够绘制平底三角形和平顶三角形的光栅化函数。这样，可以编写一个通用的光栅化函数，它将不是平顶或平底的三角形分割成两个三角形，然后对这两个三角形分别调用相应的三角形光栅化函数，如图 8.18 所示。在我们的函数库中，包含这个简单的三角形光栅化函数的多种版本，它们的原型如下：

```
// 8 位光栅化函数
void Draw_Triangle_2D(int x1,int y1, // 顶点 1
                      int x2,int y2, // 顶点 2
                      int x3,int y3, // 顶点 3
                      int color,     // 多边形的颜色索引
                      UCHAR *dest_buffer, // 目标缓存
                      int mempitch); // 缓存的跨距

// 基于顶点数的 8 位光栅化函数
void Draw_TriangleFP_2D(int x1,int y1, // 顶点 1
                        int x2,int y2, // 顶点 2
                        int x3,int y3, // 顶点 3
                        int color,     // 多边形的颜色索引
                        UCHAR *dest_buffer, // 目标缓存
                        int mempitch); // 缓存的跨距
// 16 位光栅化函数
void Draw_Triangle_2D16(int x1,int y1, // 顶点 1
                        int x2,int y2, // 顶点 2
                        int x3,int y3, // 顶点 3
                        int color,     // 多边形的 RGB 颜色
                        UCHAR *dest_buffer, // 目标缓存
                        int mempitch); // 缓存的跨距
```

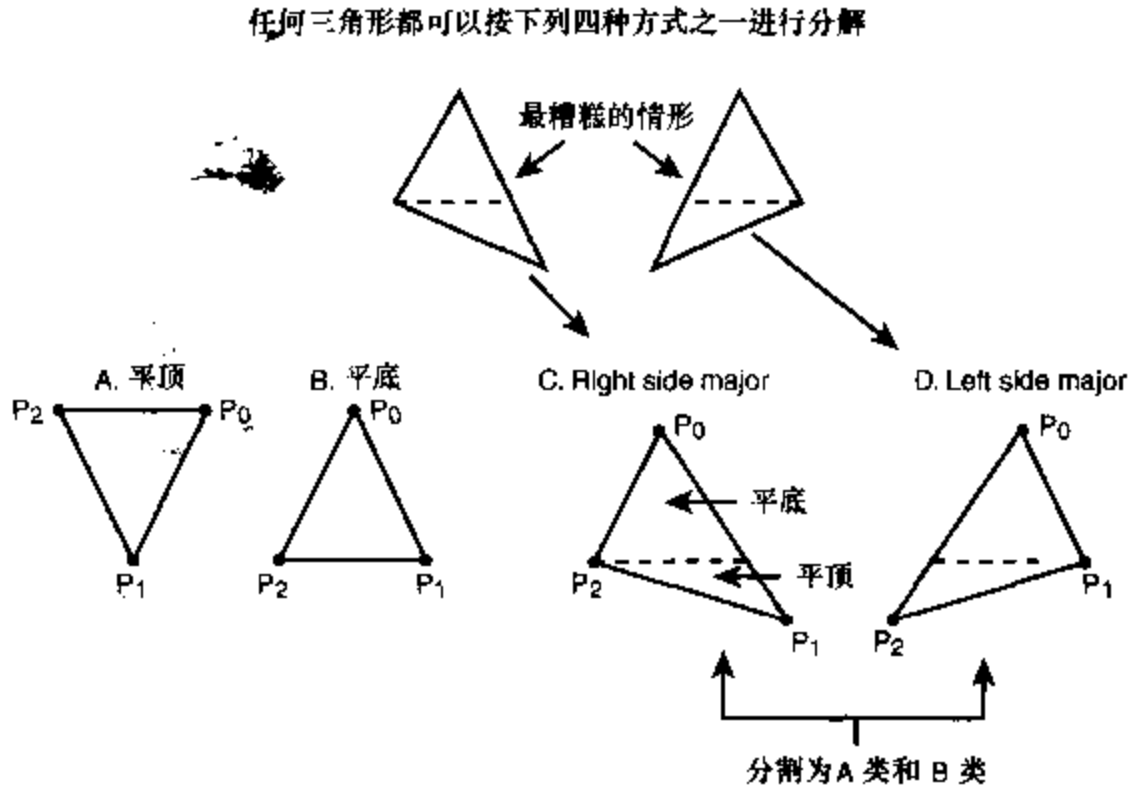


图 8.18　对三角形进行光栅化时可能出现的各种分解方式

它们都可以在 T3DLIB1.CPP|H 中找到。在本章中，新增的函数都包含在 T3DLIB6.CPP|H 中。

有了绘制实心三角形的函数后，接下来将这种功能加入到 3D 引擎中。为此，只需对绘制线框物体和渲染列表的函数进行修改，使之调用三角形渲染函数，仅此而已。下面完整地列出了用于绘制实心物体和多边形列表的新函数（读者对其中的参数应该不陌生）：

```
// 绘制实心 8 位物体
void Draw_OBJECT4DV1_Solid(OBJECT4DV1_PTR obj,
                           UCHAR *video_buffer, int lpitch);

// 绘制实心 16 位物体
void Draw_OBJECT4DV1_Solid16(OBJECT4DV1_PTR obj,
                             UCHAR *video_buffer, int lpitch);

// 绘制实心 8 位渲染列表
void Draw_RENDERLIST4DV1_Solid(RENDERLIST4DV1_PTR rend_list,
                               UCHAR *video_buffer, int lpitch);

// 绘制实心 16 位渲染列表
void Draw_RENDERLIST4DV1_Solid16(RENDERLIST4DV1_PTR rend_list,
                                 UCHAR *video_buffer, int lpitch);
```

首先来看使用 8 位颜色绘制 OBJECT4DV1 的函数，其代码如下：

```
void Draw_OBJECT4DV1_Solid(OBJECT4DV1_PTR obj,
                           UCHAR *video_buffer, int lpitch)
{
// 这个函数使用实心和 8 位模式将物体渲染到屏幕上
// 没有考虑隐藏面消除等问题
// 这个函数是一种渲染物体的简单方式，不要求将物体转换为多边形
// 它假设坐标为屏幕坐标，并执行 2D 裁剪
// 遍历物体的多边形列表，绘制其中的每个多边形
for (int poly=0; poly < obj->num_polys; poly++)
    {
    // 当且仅当多边形没有被裁剪掉、没有被剔除掉、处于活动状态且可见时才渲染它
    // 在线框引擎中，背面的概念无关紧要
    if (!(obj->plist[poly].state & POLY4DV1_STATE_ACTIVE) ||
       (obj->plist[poly].state & POLY4DV1_STATE_CLIPPED ) ||
       (obj->plist[poly].state & POLY4DV1_STATE_BACKFACE) )
      continue; // 进入下一个多边形

    // 提取指向主列表的顶点索引
    // 多边形不是自包含的，而是基于物体的顶点列表的
    int vindex_0 = obj->plist[poly].vert[0];
    int vindex_1 = obj->plist[poly].vert[1];
    int vindex_2 = obj->plist[poly].vert[2];

    // 绘制三角形
    Draw_Triangle_2D(obj->vlist_trans[ vindex_0 ].x, obj->vlist_trans[ vindex_0 ].y,
    obj->vlist_trans[ vindex_1 ].x, obj->vlist_trans[ vindex_1 ].y,
    obj->vlist_trans[ vindex_2 ].x, obj->vlist_trans[ vindex_2 ].y,
    obj->plist[poly].color, video_buffer, lpitch);
    } // end for poly
} // end Draw_OBJECT4DV1_Solid
```

正如读者看到的，只需提取顶点索引，并利用它们将三角形 3 个顶点的屏幕坐标 x 和 y 传递给三角形渲染函数即可。

提示：读者可能会问，“如何处理裁剪呢”。现在，裁剪仍然是在屏幕空间中进行的，但 3D 物体剔除除外。因此，将多边形传递给渲染流水线时，屏幕光栅化函数将对每个像素进行裁剪，就现在而言，这是可行的。

接下来看一看使用 16 位颜色模式和实心模式绘制渲染列表的函数：

```
void Draw_RENDERLIST4DV1_Solid16(RENDERLIST4DV1_PTR rend_list,
UCHAR *video_buffer, int lpitch)
{
```

```
// 这个函数"执行"渲染列表
// 即使用实心 16 位模式绘制渲染列表中所有多边形
// 这里没有对多边形进行排序，但后面将这样做以避免绘制隐藏面
// 另外，我们让函数来判断位深，并调用相应的光栅化函数

// 绘制渲染列表
for (int poly=0; poly < rend_list->num_polys; poly++)
    {
    // 当且仅当多边形没有被裁剪掉、没有被剔除掉、处于活动状态且可见时才渲染它
    // 在线框引擎中，背面的概念无关紧要
   if (!(rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_ACTIVE) ||
      (rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_CLIPPED ) ||
      (rend_list->poly_ptrs[poly]->state & POLY4DV1_STATE_BACKFACE) )
   continue; // 进入下一个多边形

    // 绘制三角形
    Draw_Triangle_2D16(rend_list->poly_ptrs[poly]->tvlist[0].x,
                       rend_list->poly_ptrs[poly]->tvlist[0].y,
                       rend_list->poly_ptrs[poly]->tvlist[1].x,
                       rend_list->poly_ptrs[poly]->tvlist[1].y,
                       rend_list->poly_ptrs[poly]->tvlist[2].x,
                       rend_list->poly_ptrs[poly]->tvlist[2].y,
                       rend_list->poly_ptrs[poly]->color, video_buffer, lpitch);
    } // end for poly
} // end Draw_RENDERLIST4DV1_Solid16
```

正如读者看到的，这个函数只有几行代码。

现在，我们需要将光照和渲染关联起来，以便能够根据光源和材质来绘制多边形。

8.2.1　为光照做准备

有了材质、光源及其数学模型后，来看一下新的 3D 流水线，看看是否一切准备就绪（至少以抽象的方式准备就绪，因为本章要实现的东西很少）。新的流水线如图 8.19 所示，其中新增了光照阶段。光照阶段位于背面和物体消除阶段之后，因此是在世界空间中执行的；然而，光照也可以在相机空间中进行。惟一不能在其中执行光照的地方是透视空间和屏幕空间，因为此时没有 3D 信息，游戏空间已被投影到平面上。

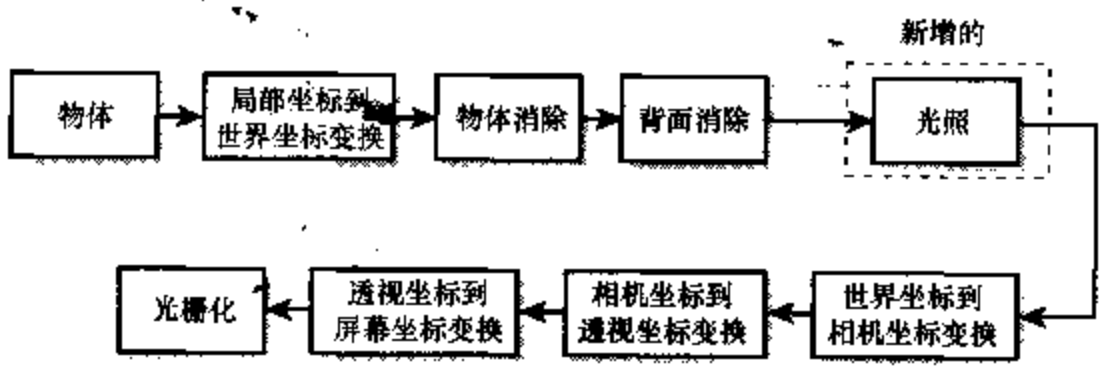


图 8.19　包含光照阶段的 3D 流水线

这里之所以提醒读者注意光照阶段在 3D 流水线中的位置，是因为工作在什么地方和什么时候做现在已经变得很重要。例如，光照计算是基于大量的面法线的，这要求我们必须根据预先计算好的法线来执行光照计算：要么在背面消除阶段执行光照计算，要么在背面消除阶段将法线存储起来，以便在光照计算阶段使用它们。

正如读者看到的，需要考虑的问题很多。总之，我们想编写出相当优化的代码，同时又要避免失去通用性，否则还不如使用汇编语言来编程。

对多边形进行光照计算之前，有必要讨论以下材质和光源。这涉及的问题很多。首先是使用的模型文件格式。PLG/PLX 格式对材质的支持不多，事实上，它根本不支持。它确实支持固定（constant）着色、恒定（flat）着色、Gouraud 着色和 Phong 着色等标记，但不支持任何描述材质的反射系数等内容的信息。另外，我们的引擎还不支持光源，因此必须采用某种方式添加这种支持。秉承作者的“简单”哲学，作者不知道下一章会考虑什么，但确实知道我们使用的光照模型和材质将非常简单；否则我们只能等待奔腾 10 的面世。下面介绍第 1 版材质定义，可将其用于多边形表面。虽然它用得很少，但还是相当完备的。

8.2.2 定义材质

正如前面讨论的，材质有很多属性，如反射系数和颜色。另外，对材质进行抽象化时，需要考虑要应用的着色方式、纹理图等内容。当前，我们在多边形属性字段中，使用标记来记录很多信息：

```
// 多边形和多边形面的属性
#define POLY4DV1_ATTR_2SIDED                0x0001
#define POLY4DV1_ATTR_TRANSPARENT           0x0002
#define POLY4DV1_ATTR_8BITCOLOR             0x0004
#define POLY4DV1_ATTR_RGB16                 0x0008
#define POLY4DV1_ATTR_RGB24                 0x0010

#define POLY4DV1_ATTR_SHADE_MODE_PURE       0x0020
#define POLY4DV1_ATTR_SHADE_MODE_CONSTANT   0x0020 // 别名
#define POLY4DV1_ATTR_SHADE_MODE_FLAT       0x0040
#define POLY4DV1_ATTR_SHADE_MODE_GOURAUD    0x0080
#define POLY4DV1_ATTR_SHADE_MODE_PHONG      0x0100
#define POLY4DV1_ATTR_SHADE_MODE_FASTPHONG  0x0100 // 别名
#define POLY4DV1_ATTR_SHADE_MODE_TEXTURE    0x0200

// 多边形和多边形面的状态
#define POLY4DV1_STATE_ACTIVE               0x0001
#define POLY4DV1_STATE_CLIPPED              0x0002
#define POLY4DV1_STATE_BACKFACE             0x0004
```

现在，我们要将这些信息交给材质去处理：不在每个多边形中存储这些数据，而是定义大量的材质，在渲染多边形时应用这些材质。图 8.20 说明了材质和多边形之间的关系。这样，我们可以使用多种材质来绘制多边形。在渲染期间，引擎根据多边形中存储的 ID 或指针找到相应的材质，并根据该材质而不是多边形的内部信息来渲染多边形。

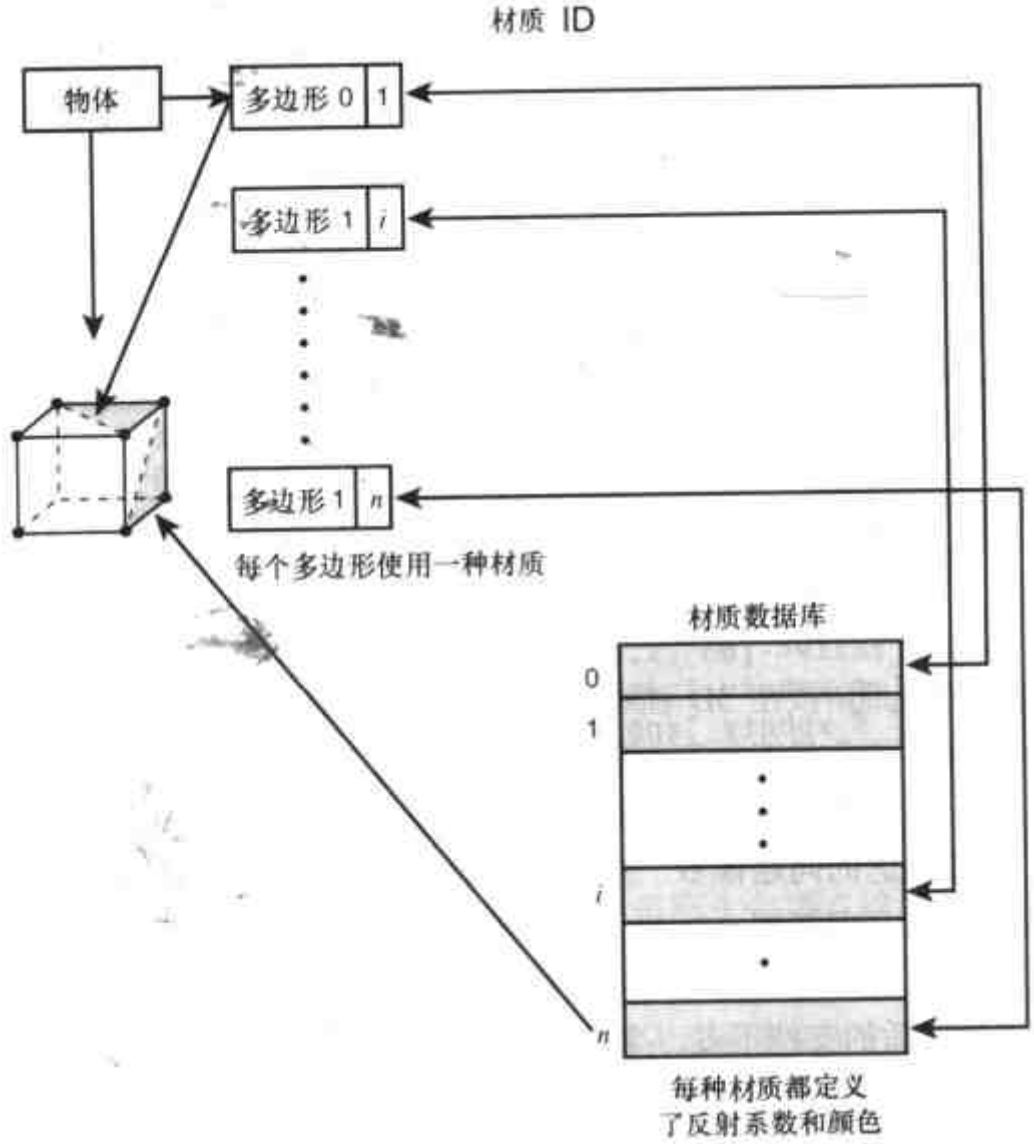


图 8.20　材质和多边形之间的关系

下面是作者创建的第一版材质结构。当然，我们可能对其进行修改，本章很少使用它，但我们必须按这样的方式思考。下面是用于材质结构的#define。

```
// 材质属性，尽可能使其与多边形属性一致
#define MATV1_ATTR_2SIDED                    0x0001
#define MATV1_ATTR_TRANSPARENT               0x0002
#define MATV1_ATTR_8BITCOLOR                 0x0004
#define MATV1_ATTR_RGB16                     0x0008
#define MATV1_ATTR_RGB24                     0x0010

#define MATV1_ATTR_SHADE_MODE_CONSTANT       0x0020
#define MATV1_ATTR_SHADE_MODE_EMMISIVE       0x0020 // 别名
#define MATV1_ATTR_SHADE_MODE_FLAT           0x0040
#define MATV1_ATTR_SHADE_MODE_GOURAUD        0x0080
#define MATV1_ATTR_SHADE_MODE_FASTPHONG      0x0100
#define MATV1_ATTR_SHADE_MODE_TEXTURE        0x0200


// 材质状态
#define MATV1_STATE_ACTIVE                   0x0001

// defines for material system
#define MAX_MATERIALS                        256
```

下面是材质数据结构：

```
// 第 1 版材质数据结构
typedef struct MATV1_TYP
{
int state;          // 材质的状态
int id;             // 材质 id，指向材质数组的索引
char name[64];      // 材质名称
int  attr;          // 属性：着色模式、着色方法、环境、纹理以及其他特殊标记

RGBAV1 color;                // 材质颜色
float ka, kd, ks, power;    //对环境光、散射光和镜面反射光的反射系数和镜面反射指数
                            // 它们是独立的标量，很多建模程序都采用这种格式
RGBAV1 ra, rd, rs;          // 预先计算得到的颜色和反射系数的积
                            // 旨在使该数据结构与我们的光照模型更为一致
char texture_file[80];     // 包含纹理的文件的位置
BITMAP texture;           // 纹理图

} MATV1, *MATV1_PTR;
```

其中没有什么奇特的内容。作者喜欢在所有结构中都包含 state、id 和 attr 等字段，但读者应将注意力放在其他字段上，尤其是 color 字段。实际上，它是一个共用体（union），以不同格式存储 r、g、b 和 alpha 值。

```
// RGBA 值
typedef struct RGBAV1_TYPE
{
union
    {
    int rgba;                       // 压缩格式
    UCHAR rgba_M[4];                // 数组格式
    struct {  UCHAR a,b,g,r;  };    // 显式名称格式
    }; // end union

} RGBV1, *RGBV1_PTR;
```

下面的宏用于创建 32 位的 RGBA 值：

```
// 生成 8.8.8.a 格式(alpha 占 8 位)的 32 位颜色值
#define _RGBA32BIT(r,g,b,a) ((a) + ((b) << 8) + ((g) << 16) + (r << 24))
```

注意：我们已经有 macro_RGB32BIT(a, r, g, b)，它创建与 DirectX 兼容的 32 位字，但作者不喜欢 alpha 值位于最前面，因为作者发现，在有些情况下，将 alpha 值存储在最后一个字节中将会有所帮助，因此使用 RGBA 格式。

另外，这个结构中还包含反射系数 ka、kd 和 ks 和镜面反射指数 power。很多建模程序都使用 K 来表

示反射系数，这里遵循了这种约定。但本章前面推导光照模型时，使用的是 R。另外，在推导光照模型时，颜色和反射系数被合而为一，但在这个数据结构中，它们是分开的。然而，为使模型和数据结构更为一致，数据结构中包含字段 ra、rd 和 rs，它们是颜色和反射系数的乘积。可能需要其中的一种格式、另一种格式或两种都要，因此这里包含了这两种格式。下面是一个使用标量 ka、kd 和 ks 以及 ARGB 颜色来计算 ra、rd 和 rs 的例子（摘自本章后面将介绍的 Caligari trueSpace 物体加载函数）：

```
// 预先计算材质的反射系数
for (int rgb_index=0; rgb_index < 3; rgb_index++)
    {
    // 环境光反射系统
    materials[material_index+num_materials].ra.rgba_M[rgb_index] =
     ( (UCHAR)(materials[material_index + num_materials].ka *
     (float)materials[material_index +
      num_materials].color.rgba_M[rgb_index] + 0.5) );
    // 散射光反射系统
    materials[material_index+num_materials].rd.rgba_M[rgb_index] =
     ( (UCHAR)(materials[material_index + num_materials].kd *
     (float)materials[material_index +
      num_materials].color.rgba_M[rgb_index] + 0.5) );

     // 镜面光反射系统
     materials[material_index+num_materials].rs.rgba_M[rgb_index] =
     ( (UCHAR)(materials[material_index + num_materials].ks *
     (float)materials[material_index +
     num_materials].color.rgba_M[rgb_index] + 0.5) );

     } // end for rgb_index
```

最后，该数据结构还支持指定纹理图及其名称。

有了材质数据结构后，需要一个用于存储它们的全局材质库，如下所示：

```
MATV1 materials[MAX_MATERIALS]; // 系统中的材质
int num_materials;              // 当前的材质数
```

非常简单：只需新建材质，将其插入到材质库中，然后应用于多边形。当然，需要解决的问题是，让多边形和引擎同材质关联起来，这将在后面进行。现在，编写一个重置材质库的函数：

```
int Reset_Materials_MATV1(void)
{
// 这个函数重置所有的材质
static int first_time = 1;

// 如果这是第一次调用该函数，则重置所有的材质
if (first_time)
   {
   memset(materials, 0, MAX_MATERIALS*sizeof(MATV1));
   first_time = 0;
   } // end if

// 遍历所有的材质，释放其纹理（如果有的话）
for (int curr_matt = 0; curr_matt < MAX_MATERIALS; curr_matt++)
    {
    // 不管材质是否处于活动状态，都释放与之相关联的纹理图
    Destroy_Bitmap(&materials[curr_matt].texture);
    // 现在可以安全地重置材质
    memset(&materials[curr_matt], 0, sizeof(MATV1));
    } // end if

return(1);
} // end Reset_Materials_MATV1
```

要重置材质库，只需这样调用上述函数即可：

```
Reset_Materials_MATV1();
```

在实际使用材质和支持材质的模型格式之前，作者不知道是否将编写一个设置材质的辅助函数，因此下面来看一个手工设置材质的例子。假设希望材质反射 100%的环境光和散射光，颜色为纯蓝，使用恒定着色，没有纹理：

```
// 设置材质数组中的第一个材质
MATV1_PTR m = &Materials[0];
// 重置材质
memset(m, sizeof(MATV1);

m->state = MATV1_ACTIVE;   // 材质状态

m->id = 0;                 // 材质 ID

strcpy(m->name, "blue mat 1"); // 材质名称
m->attr = MATV1_ATTR_2SIDED | MATV1_ATTR_16BIT
| MATV1_ATTR_FLAT; // 属性

m->color.rgba = _RGBA32BIT(0,0,255, 255); // 设置为蓝色
m->ka    = 1.0;  // 将环境光反射系数设置为 1.0
m->kd    = 1.0;  // 将散射光反射系数设置为 1.0
m->ks    = 1.0;  // 将镜面光反射系数设置为 1.0
m->power = 0.0;  // 没有使用，将其设置为 0.0

// 这些是实际的 RGBA 颜色，因此将颜色乘以反射系数
m->ra.b = (UCHAR)(m->ka * (float)m->color.b +0.5);
m->rd.b = (UCHAR)(m->ka * (float)m->color.b +0.5);
m->rs.rgba = 0; // 不反射镜面光
```

显然，需要编写一个辅助函数来完成这项任务，但由于不知道将如何定义材质——手工定义还是通过加载物体来定义，还是将这项工作推迟到以后去完成吧。

8.2.3　定义光源

光源的实现有些类似于在没有奴佛卡因（一种局部麻醉剂——译者注）的情况下做牙根管填充手术——非常棘手。一切都好像充满希望，但当您真正去做时，却发现没有足够的处理器周期来计算大量的点积和三角函数。

这可能过于夸张，但读者应该明白了作者的意思。一般而言，优秀的光照引擎应该支持无穷多的各种类型的光源，而光照计算应针对每个像素，并考虑所有的材质属性——实际情况并非如此。即使是最高级的 3D 加速器也只支持不多的光源（大多数情况下为 8～16 个）。然而，有办法避免这种限制，例如使用光源映射，这将在后面介绍。就现在而言，读者需要牢记的是，我们不能执行完美的像素光照计算，同时通常只支持无穷远光源和点光源。另外，所有的光照计算都只在三角形顶点上进行，然后使用插值来平滑表面的光照（Gouraud/Phong 着色）。

从上面的介绍可知，读者使用软件编写像素 shader 的美梦已经破灭。下面定义一种基本的光源数据结构，它能够表示本章前面介绍过的 3 种光源。

下面是用于光源数据结构（1.0 版）的#define。注意，这里将聚光灯分为两种，因此后面可以使用标准的和简化的聚光灯模型。

```
// 有关光源的常量
#define LIGHTV1_ATTR_AMBIENT      0x0001    // 环境光源
#define LIGHTV1_ATTR_INFINITE     0x0002    // 无穷远光源
#define LIGHTV1_ATTR_POINT        0x0004    // 点光源
#define LIGHTV1_ATTR_SPOTLIGHT1   0x0008    // 1 类（简单）聚光灯
#define LIGHTV1_ATTR_SPOTLIGHT2   0x0010    // 2 类（复杂）聚光灯

#define LIGHTV1_STATE_ON          1         // 光源打开
#define LIGHTV1_STATE_OFF         0         // 光源关闭

#define MAX_LIGHTS                8         // 最多支持多少个光源
```

基本上，我们需要选择光源的类型，并能够将其状态设置为开或关。下面是光源数据结构：

```
// 光源数据结构 1.0 版
typedef struct LIGHTV1_TYP
```

```
{
int state;          // 光源状态
int id;             // 光源 id
int attr;           // 光源类型及其他属性

RGBAV1 c_ambient;   // 环境光强度
RGBAV1 c_diffuse;   // 散射光强度
RGBAV1 c_specular;  // 镜面反射光强度
POINT4D  pos;       // 光源位置
VECTOR4D dir;       // 光源方向
float kc, kl, kq;   // 衰减因子
float spot_inner;   // 聚光灯内锥角
float spot_outer;   // 聚光灯外锥角

float pf;           // 聚光灯指数因子

} LIGHTV1, *LIGHTV1_PTR;
```

下面是全局光源数组，它与材质数组极其类似：

```
LIGHTV1 lights[MAX_LIGHTS];  // 光源数组
int num_lights;              // 当前的光源数
```

在实际渲染期间，可能使用材质或光源，也可能两者都使用，还可能两者都不使用，这取决于要使用引擎做什么。现在，我们只是创建以后可能要用到的东西的模型，以免在需要它们时做蠢事。设计 3D 引擎时，预先进行规划至关重要，因为这样在编写引擎时知道各个组件的运行方式。如果等到必须加入某种组件时才去考虑，将为此付出代价。一个这样的典型例子是网络支持，如果等到引擎已经编写好后才想起要加入网络支持，那将是无法实现的。

谈完了设计原则，下面创建一些光源。不同于材质，对于光源需要考虑的问题并不多。光源要么存在要么不存在，使用多少光源以及对光源模拟到什么程度由引擎决定，但我们可以编写一个这样的辅助函数：它创建的光源无需经过很多修改就可使用。基本上，只需根据要创建的光源类型，传递类型、位置和方向等参数即可。另外，由于将使用的光源数不多（4～8 个），因此将支持把数组中的光源打开，然后在光照阶段遍历光源数组，并对打开的光源进行处理。首先，编写一个内务处理函数，它初始化所有光源，将它们关闭。

```
int Reset_Lights_LIGHTV1(void)
{
// 这个函数重置系统中所有的光源
static int first_time = 1;

memset(lights, 0, MAX_LIGHTS*sizeof(LIGHTV1));
// 重置光源数
num_lights = 0;

first_time = 0;

// 成功返回
return(1);

} // end Reset_Lights_LIGHTV1
```

这样，在引擎/游戏的初始化部分，只需调用 Reset_Lights_LIGHTV1()。下面是创建光源的辅助函数：

```
int Init_Light_LIGHTV1(
    int           index,     // 要创建的光源的索引（0 到 MAX_LIGHTS-1）
    int           _state,    // 光源状态
    int           _attr,     // 光源类型及其他属性
    RGBAV1        _c_ambient, // 环境光强度
    RGBAV1        _c_diffuse, // 散射光强度
    RGBAV1        _c_specular,// 镜面反射光强度
    POINT4D_PTR   _pos,      // 光源位置
    VECTOR4D_PTR  _dir,      // 光源方向
    float kc, kl, kq,        // 衰减因子
    float         _spot_inner,// 聚光灯内锥角
    float         _spot_outer,// 聚光灯外锥角
    float         _pf)       // 聚光灯指数因子
```

```
{
// 这个函数根据传入的参数初始化光源
// 调用该函数时，为确保创建的光源有效将不需要的参数值设置为 0
if (index < 0 || index >= MAX_LIGHTS)
   return(0);

// 初始化光源
lights[index].state       = _state;         // 光源状态
lights[index].id          = index;          // 光源 id
lights[index].attr        = _attr;          // 光源类型及其他属性
lights[index].c_ambient   = _c_ambient;     // 环境光强度
lights[index].c_diffuse   = _c_diffuse;     // 散射光强度
lights[index].c_specular  = _c_specular;    // 镜面反射光强度
lights[index].kc  = _kc;                    // 常量、线性和二次衰减因子
lights[index].kl  = _kl;
lights[index].kq  = _kq;

if (pos)
    VECTOR4D_COPY(&lights[index].pos, _pos;  // 光源位置
if (dir)
    {
    VECTOR4D_COPY(&lights[index].dir, _dir;  //光源方向
    // 归一化
    VECTOR4D_Normalize(&lights[index].dir);
    } // end if

lights[index].spot_inner    = _spot_inner; // 聚光灯内锥角
lights[index].spot_outer    = _spot_outer; // 聚光灯外锥角
lights[index].pf          = _pf;           // 聚光灯指数因子

// 返回光源索引
return(index);

} // end Create_Light_LIGHTV1
```

这个函数执行错误检查后，将传入的参数值复制到光源结构中，并返回创建的光源的索引。下面使用这个函数来创建几个光源。

1. 创建环境光源

每个场景都可能有环境光源，实际上，我们的第一个实心引擎只使用环境光源。下面是一个创建环境光源的例子：

```
ambient_light = Init_Light_LIGHTV1(0,  // 对于环境光源使用索引 0
    LIGHTV1_STATE_ON,                  // 打开光源
    LIGHTV1_ATTR_AMBIENT,              // 环境光源
    _RGBA32BIT(255,255,255,0), 0, 0,   // 纯白色环境光
    NULL, NULL,                        // 不需要指定位置和方向
    0,0,0,                             // 不需要指定衰减因子
    0,0,0);                            // 聚光灯属性不适用
```

开启光源后，可以通过手工访问状态变量来将其关闭，如下所示：

```
Lights[ambient_light].state = LIGHTV1_STATE_OFF;
```

2. 创建无穷远光源（定向光源）

接下来创建一个类似于太阳的无穷远散射光光源。因此将颜色设置为黄色，将其方向设置为指向+y 轴，如图 8.21 所示（注意，所有的方向向量都必须是单位向量）。

```
VECTOR4D sun_dir = {0, -1, 0, 0};

sun_light = Init_Light_LIGHTV1(1,       // 索引为 1
            LIGHTV1_STATE_ON,           // 打开光源
            LIGHTV1_ATTR_INFINITE,      //  无穷远光源
            0, RGBA32BIT(255,255,0,0), 0, // 纯黄色散射光
            NULL, &sun_dir, // 不需要指定位置，方向为(0,-1,0)
            0,0,0,          // 不需要指定衰减因子
```

```
0,0,0);                    // 聚光灯属性，不适用
```

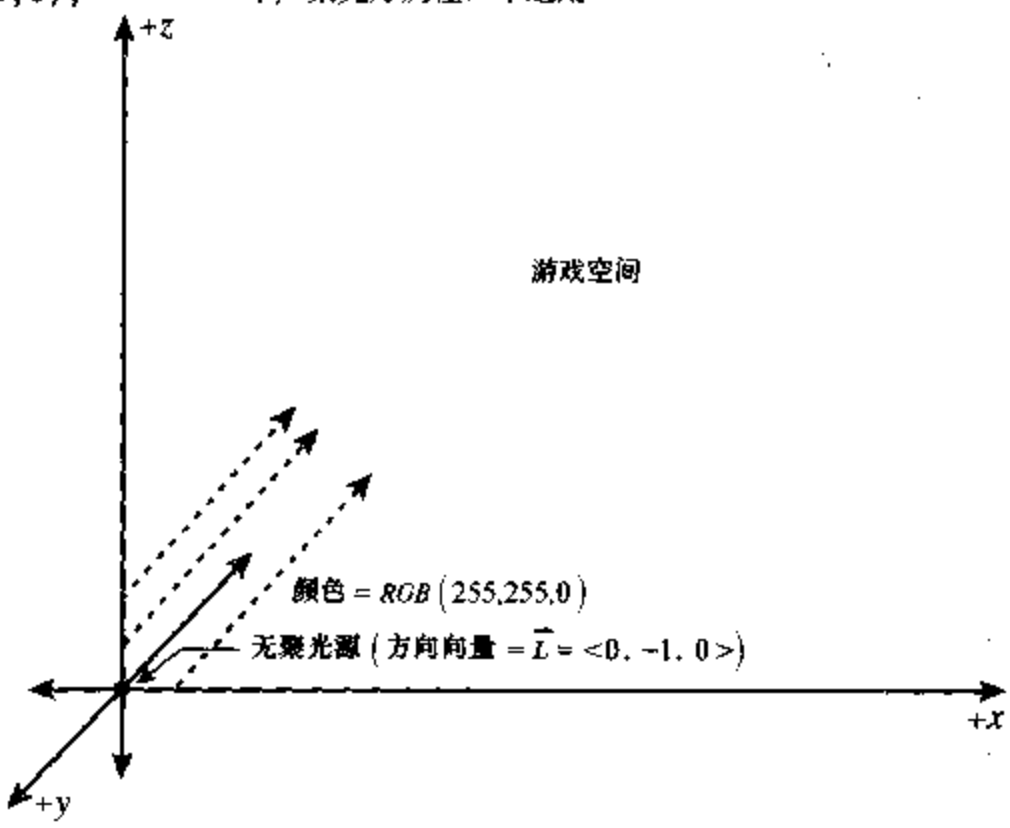


图 8.21　无穷远光源

3. 创建点光源

点光源有位置，这里将其放置在+y 轴上离原点 10 000 个单位处，其他参数与前一个光源相同，如图 8.22 所示。

```
VECTOR4D sun_pos = {0, 10000, 0, 0};

sun_light = Init_Light_LIGHTV1(1,              // 索引为 1
LIGHTV1_STATE_ON,                  // 打开光源
LIGHTV1_ATTR_POINT,                //  点光源
0, RGBA32BIT(255,255,0,0), 0,  // 纯黄色散射光
&sun_pos, NULL,  // 位于 y 轴上，不需要指定方向
0,1,0,           // 线性衰减因子为 1
0,0,0);          // 聚光灯信息，不适用
```

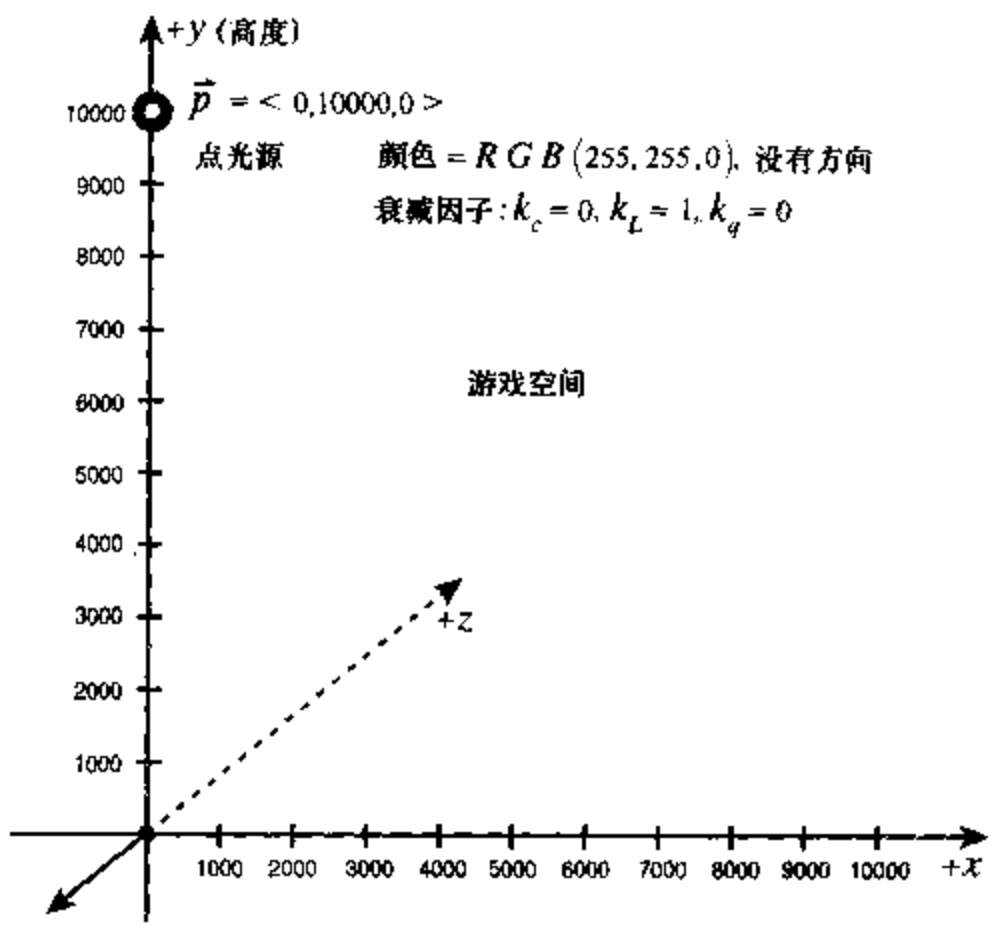


图 8.22　聚光灯

4. 创建聚光灯

最后，创建一个聚光灯。从其数学运算的复杂性可知，实时地渲染时，无法使用完整的聚光灯模型。另外，只能在顶点上执行聚光灯光照计算，因为对每个像素执行这种计算是极其愚蠢的。因此，需要将物体划分成大量的多边形。这并不是说不能编写出使用聚光灯的演示程序，但在游戏中，这是无法实现的，因为速度太慢。但读者也不要失望，可以使用诸如光源映射等技术来创建聚光灯，这将在本书后面介绍。

无论如何，我们来定义一个聚光灯，其位置 **p** 为(1000，1000，−1000)，指向原点 **o**(0，0，0)，**v** = <−1，−1，1>，强度为最大强度的 50%，颜色为白色，即 RGB =(128，128，128)。它发出的是散射光，内锥（本影）角为 30 度，外锥（半影）角为 60 度，指数因子为 1.0。参见图 8.23，代码如下：

```
VECTOR4D spot_pos = {1000,1000,-1000, 0};
VECTOR4D spot_dir = {-1, -1, 1,0};
float umbra = 30, penumbra = 60, falloff = 1.0;

int spot_light = Init_Light_LIGHTV1(5,// 光源索引为 5
                 LIGHTV1_STATE_ON,    // 打开光源
                 LIGHTV1_SPOTLIGHT1, //  1 类聚光灯
     _           _0, RGBA32BIT(128,128,128,0), 0, // 强度为 50%的白色散射光
                 &spot_pos, &spot_dir,         // 聚光灯位置
                 umbra,penumbra,falloff);      // 聚光灯信息
```

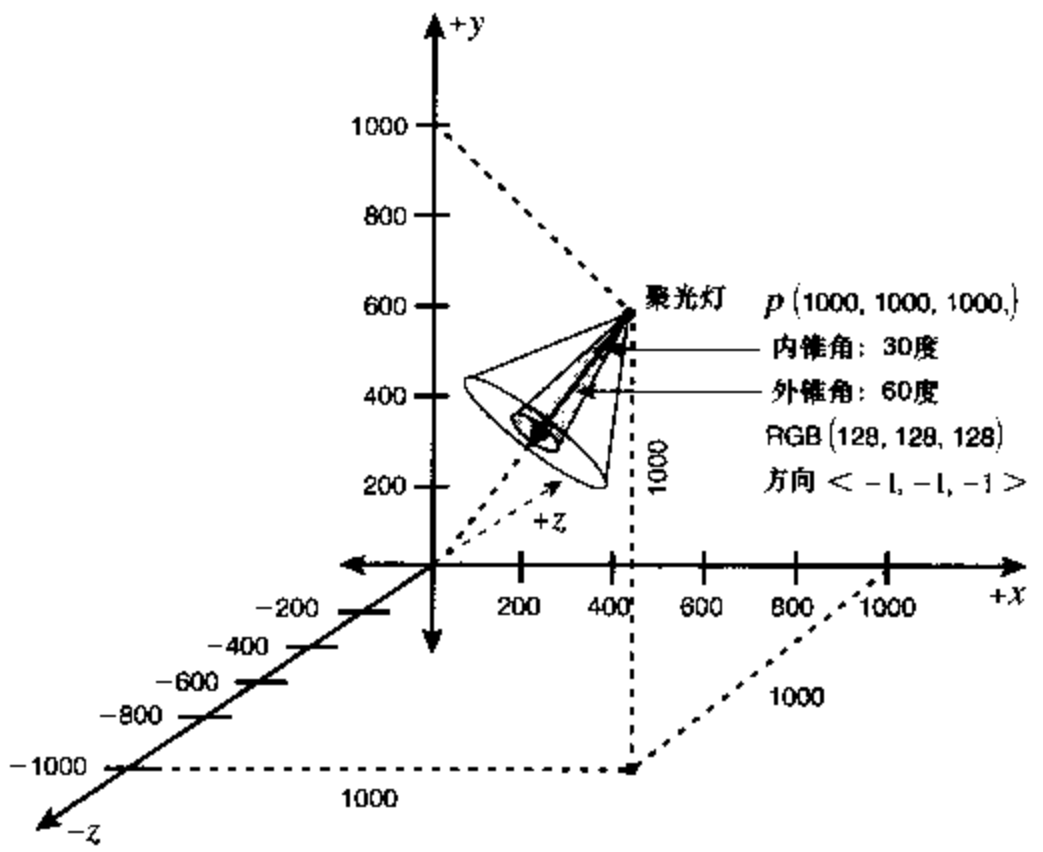


图 8.23　聚光灯

8.3　真实世界中的着色

我们几乎为实际绘制做好了准备。有趣的是，本章介绍的内容有 50%要等到后面才会被使用。无论如何，这里还是讨论一下在 8/16 位图形模式下着色的实现和优化。在 8 位模式的速度为 16 位模式的 3 倍，因为在这种模式下，无需处理 3 个 RGB 分量，而只需处理单个值——亮度或辉度。

8.3.1　16 位着色

着色首先涉及的是材质和光源。本章前面的模型是基于 RGB 空间的。例如，材质的反射系统是 RGB 值，光源的颜色是 RGB 值，等等。因此，在大多数情况下，可能按本章开头概述的那样来执行光照计算。当然最终的实现可能进行了简化（使用假设条件）。

例如，可能将所有光源都设置为白色的，以提高速度；还可能假定所有物体只受环境光、散射光和发射光属性的影响。然而，不管我们怎样做，都必须在某个时候计算多边形、顶点或像素的最终颜色，这意味着必须执行一些数学运算，但作者将尽可能做较少数学运算。需要指出的是，基于每个多边形或顶点执行某种操作没有什么大不了的，但基于每个像素执行任何操作都将是让人无法承受的。

接下来的问题是，对于数学运算，将使用什么样的格式？为什么？我们将使用 16 位 RGB 模式，这意味着需要在某个时候将计算结果转换为 16 位 RGB 模式。那么应该在 16 位模式下执行光照计算还是使用一个更通用的方式避免转换？这是读者必须考虑的问题。对于这个问题，惟一的答案是，使用速度最快的方法。

8.3.2 8 位着色

从数学上说，16 位着色比 8 位着色更容易实现。原因在于，RGB 颜色空间中的强度和屏幕上像素的强度之间存储一一对应的关系。例如，多边形为纯蓝色（RGB=(0，0，255)），并反射散射光和环境光。现在假设活动光源的光线强度为 0.5，环境光强度为 0.2。

考虑这种问题的方式有两种。如果材质颜色为蓝色，环境光为红色，则完全在 RGB 空间中工作时，将看不到任何东西，其原因如下：

```
Final_pixelrgb = materialrgb*ambientrgb
= (0,0,255)*(1.0,0,0) = (0,0,0)
```

然而，如果进行简化，假设所有光源都发射某种强度的纯白光，可以简化数学运算。在这种情况下，将能看到多边形。

```
Final_pixelrgb = materialrgb*ambientrgb
= (0,0,255)*(1.0,1.0,1.0) = (0,0,255)
```

注意：在这里，环境光被转换为 0～1 的浮点数，这样，调制计算的结果将位于颜色空间范围内。当然也可以使用 0～255 的格式，但必须对计算结果进行缩小。

8.3.3 一个健壮的用于 8 位模式的 RGB 模型

我们的引擎在很大程度上将使用白色光源以简化计算。现在的问题是，在上述两个 RGB 范例中，最后的像素颜色为 RGB 24 位格式或(0...1，0...1，0...1)格式，将后者转换为 RGB 24 位格式很容易。我们将 24 位 RGB 颜色值转换为 16 位的，然后在屏幕上显示。

然而，在 8 位模式下，这毫无意义。屏幕颜色不是 RGB 格式，而是颜色索引，这就是问题所在。那么，如何解决呢？如何在索引模式下执行光照计算？

答案很简单。需要一个中间步骤，将 RGB 值转换为颜色索引，然后将其显示到屏幕上。但问题是，只有 256 个颜色索引，因此计算出 RGB 颜色后，需要在调色板中找到与之最接近的颜色索引，将其作为最后的颜色，如图 8.24 所示。假设 RGB 颜色为(10，45，34)，调色板中与之最接近的颜色索引为 45，后者对应的 RGB 颜色为(10，48，40)——这将是返回的值。但这意味着绘制每一个像素时，都需要找出一个最小平方值并搜索整个调色板。为避免这一点，需要预先计算一个查找表。我们需要做的是：根据 RGB 颜色空间创建一个查找表，它将输入的 RGB 映射到一个索引。RGB 值被用作查找表的索引。

提示：使用 8 位颜色模式进行实时着色时，必须使用一个覆盖了很大的 RGB 颜色空间，且有很多着色等级（shade）的调色板。如果使用的调色板只覆盖颜色空间的很小一部分，关照处理的结果将极其丑陋。作者在位图文件 PALDATAxx.BMP|PAL 中提供的调色板是不错的选择，其效果很不错。当然所有的图形都必须使用这些颜色，因为在 8 位模式下只有一个调色板。

无论是在 RGB 8.8.8 模式还是 16 位模式（5.5.5 或 5.6.5 格式）下执行光照计算，都不能创建一个将 8.8.8 RGB 值映射到索引的表——这将占用 4GB 的空间，实在太大了。然而，可以这样做：创建一个将 16 位 RGB 值映射到 8 位索引的查找表，这只需要 64KB 内存——完全负担得起。另外，由于 16 位值可能为 5.5.5 和 5.6.5 格式，因此需要创建两个查找表，每种格式一个。现在，对于输入的 24 位 RGB 值，我们首先将

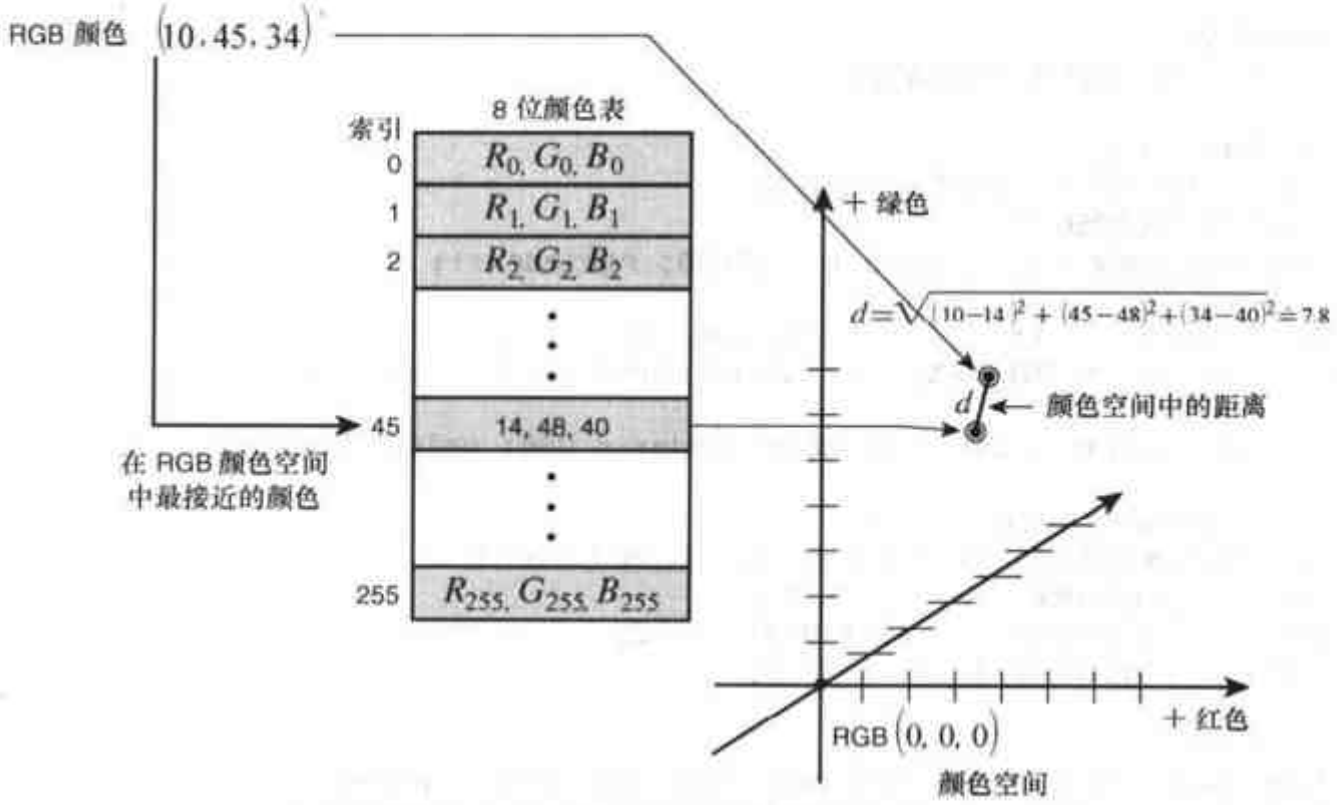


图 8.24　在调色板中查找最接近的索引

其转换为 16 位的 5.5.5 或 5.6.5 格式。然后根据这个值在查找表中找到相应的单字节索引值，该索引指向调色板中与输入值最接近的 RGB 值，如图 8.25 所示。

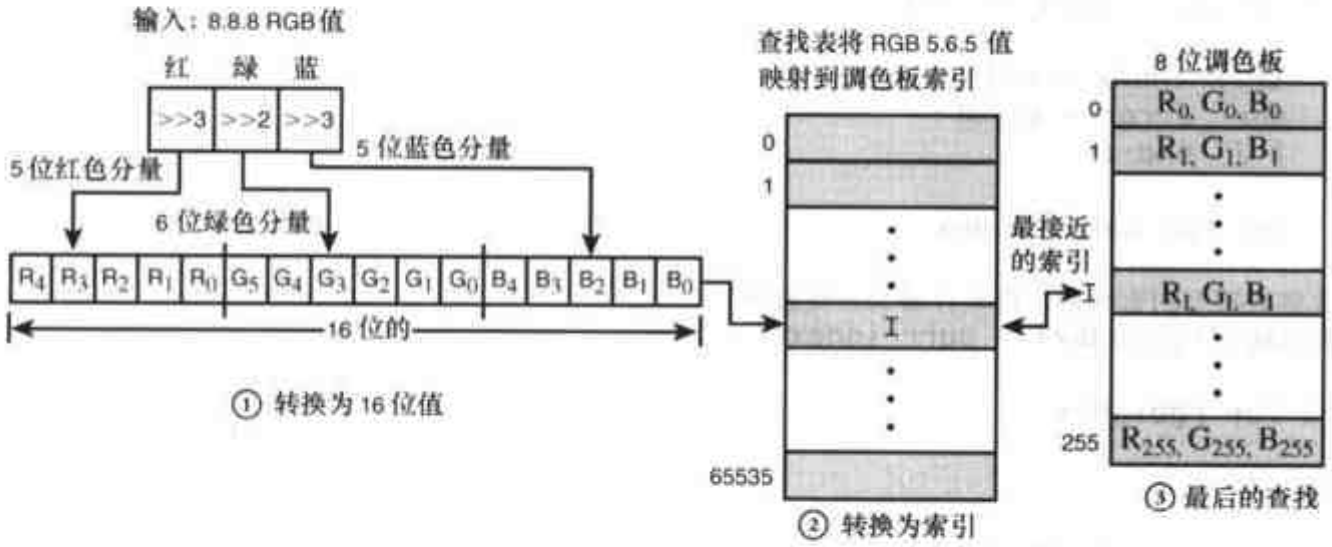


图 8.25　RGB 颜色转换查找表的结构

因此，需要编写一个这样的函数：它将 16 位格式、8 位调色板和指向查找表的指针作为参数，计算转换结果并将其存储到查找表中（可能需要对游戏世界的不同区域使用不同的调色板和查找表；否则作者将在该函数中使用一个全局查找表）。该函数的代码如下：

```
int RGB_16_8_IndexedRGB_Table_Builder(
    int rgb_format,  // RGB 格式. 5.6.5 或 5.5.5
    LPPALETTEENTRY src_palette,   // 调色板
    UCHAR *rgblookup)             // 查找表

{
// 这个函数接受 RGB 格式作为参数，其取值为 DD_PIXEL_FORMAT565 或 DD_PIXEL_FORMAT555
// 该参数指定了要为哪种 RGB 颜色格式创建查找表
// RGB 格式为 5.5.5 时，只有 32K 种颜色（不考虑第一位的值）
// 因此查找表只包含 32K 个条目
// 调用该函数时，必须提供预先分配好的查找表存储空间（参数 rgblookup）
// 这个函数不申请内存
// 该函数使用最小平方法找出 8 位调色板中与各种 16 位 RGB 颜色
// 最接近的颜色索引

// 检查参数
if (!src_palette || !rgblookup)
    return(-1);
```

```
// 判断 RGB 颜色格式
if (rgb_format==DD_PIXEL_FORMAT565)
    {
    // 总共有 64k 个条目
    // 执行循环，找出与各种颜色最接近的调色板索引
    // 总共循环 65536*256 次
    for (int rgbindex = 0; rgbindex < 65536; rgbindex++)
    {
    int  curr_index  = -1;        // 当前最接近的颜色索引
    long curr_error  = INT_MAX;   // 最接近的颜色和当前颜色之间的距离

        for (int color_index = 0; color_index < 256; color_index++)
        {
        // 从 rgbindex 中提取 r、g、b
        // 将 5.6.5 格式转换为 8.8.8 格式，因为调色板使用这种格式
        int r = (rgbindex >> 11) << 3;;
        int g = ((rgbindex >> 5) & 0x3f) << 2;
        int b = (rgbindex & 0x1f) << 3;

        // 计算距离
        long delta_red   = abs(src_palette[color_index].peRed - r);
        long delta_green = abs(src_palette[color_index].peGreen - g);
        long delta_blue  = abs(src_palette[color_index].peBlue - b);
        long error = (delta_red*delta_red) + (delta_green*delta_green) +
                     (delta_blue*delta_blue);
        // 是否更近?
        if (error < curr_error)
           {
           curr_index = color_index;
           curr_error = error;
           } // end if

     } // end for color_index

      // 找到最接近的索引后，将其存储到查找表中
      rgblookup[rgbindex] = curr_index;

    } // end for rgbindex

    } // end if
  else
  if (rgb_format==DD_PIXEL_FORMAT555)
    {
    // 总共有 32k 个条目 entries,
    // 通过循环查找最接近的索引，总共需要执行 32768*256 次循环
    for (int rgbindex = 0; rgbindex < 32768; rgbindex++)
    {
    int  curr_index  = -1;        // 当前最接近的颜色索引
    long curr_error  = INT_MAX;   // 当前最小的距离

    for (int color_index = 0; color_index < 256; color_index++)
    {
    // 从 rgbindex 中提取 R、G、B 值
    // 然后将其从 5.5.5 格式转换为 8.8.8 格式，因为调色板使用这种格式
    int r =  (rgbindex >> 10) << 3;;
    int g = ((rgbindex >> 5) & 0x1f) << 3;
    int b =  (rgbindex & 0x1f) << 3;

    // 计算距离
    long delta_red   = abs(src_palette[color_index].peRed - r);
    long delta_green = abs(src_palette[color_index].peGreen - g);
    long delta_blue  = abs(src_palette[color_index].peBlue - b);
    long error = (delta_red*delta_red) + (delta_green*delta_green) +
                 (delta_blue*delta_blue);
    // 是否更近?
    if (error < curr_error)
```

```
        {
        curr_index = color_index;
        curr_error = error;
        } // end if
      } // end for color_index

      // 找到最接近的颜色索引后，将其存储到查找表中
      rgblookup[rgbindex] = curr_index;

  } // end for rgbindex
  } // end if
  else
    return(-1); // RGB 颜色格式不对，

  // 成功返回
  return(1);

  } // end RGB_16_8_IndexedRGB_Table_Builder
```

16 位可以表示 65536 种颜色，因此我们创建一个查找表，其中包含 0～65535 对应的调色板索引。要使用这个函数，只需像下面那样调用它，当然之前应初始化 DirectDraw 并加载 8 位调色板：

```
// 申请查找表存储空间
UCHAR rgblookup = (UCHAR *)malloc(65536);

// 使用全局 RGB 格式和调色板进行调用
RGB_16_8_IndexedRGB Table_Builder(dd_pixel_format, palette, rgblookuptable);
```

该函数执行后，rgblookup[ ]将是把（5.6.5 格式或 5.5.5 格式的）16 位 RGB 颜色转换为 8 位索引的查找表。我们按通常那样执行光照计算，将结果转换为 16 位的 RGB 颜色，然后像下面这样设置像素的最终颜色：

```
final_pixel_index = rgblookup[final_pixelrgb16];
```

来看看我们有何收获。通过预先计算，我们将转换过程变换为表查找。但对于每个像素仍需要执行一次表查找，这是否可以避免呢？答案是，无论如何这一步也是避免不了的。在 8 位模式下执行光照计算时，必须将结果转换为 8 位索引。当然，可以使用一些技巧。例如，可以创建一个这样的颜色表，其中包含每种颜色的各种着色度，然后将亮度作为索引来将多边形原来的颜色（base color）转换为表中的颜色，但这仍然需要使用索引。从本质上说，如果在 RGB 空间中执行光照计算，都将遇到困难，因为计算本身必须执行，然后需要将结果转换为 8 位索引。

是否可以创建一种面向 8 位的颜色模型，以减少 16 位 RGB 中间步骤？当然，下面介绍这种模型。

8.3.4　一个简化的用于 8 位模式的强度模型

无论在哪种模式下进行着色，计算最终的 RGB 值都会让我们筋疲力尽。然而，如果对光照模型进行简化，避免涉及光源颜色和材质颜色，则可以使用极度简化的 8 位着色模型。其工作原理如下：所有多边形都有一种调色板定义的 RGB 颜色。在调色板包含的是 8 位索引和 8.8.8 格式颜色，让我们忘记所有的 RGB 问题，跳出原来的框框，以相反的方向进行光照处理。也就是说，计算调色板中每种颜色的 256 种强度，然后创建一个由 256 行组成的表，每种调色板颜色一行。每行包含 256 项，每项都是一个调色板索引，该索引对应的颜色与相应强度和颜色的组合最接近，如图 8.26 所示。

这里的关键之处在于，为何计算全部强度值呢？我们计算调色板中有的颜色的各种强度值，然后在执行光照处理时，使用颜色作为行索引，使用强度作为列索引在表中找出相应的颜色。同样，这里也使用查找表，但避免了前述 8 位模型中涉及 16 位 RGB 的中间步骤；同时假设强度值范围为 0～255，这很简单，因为即使它们的范围不是这样，总是可以对其进行缩放。

如果读者感到迷惑，请看下面的解释。前面的模型从头到尾都使用 RGB 空间，也就说是，在 RGB 空间中执行光照计算，然后使用查找表对 16 位的 RGB 值传进行转换。然而，既然知道不是在 16 位模式下，为何这样做呢？原因在于这种模型更通用。然而，如果从“调色板中有哪些颜色”的角度来考虑问题，便只需创建一个转换表，对于调色板中的每种颜色，该表中包含 256 个值，它们表示这种颜色的强度被调制过的版本。这种方法之所以可

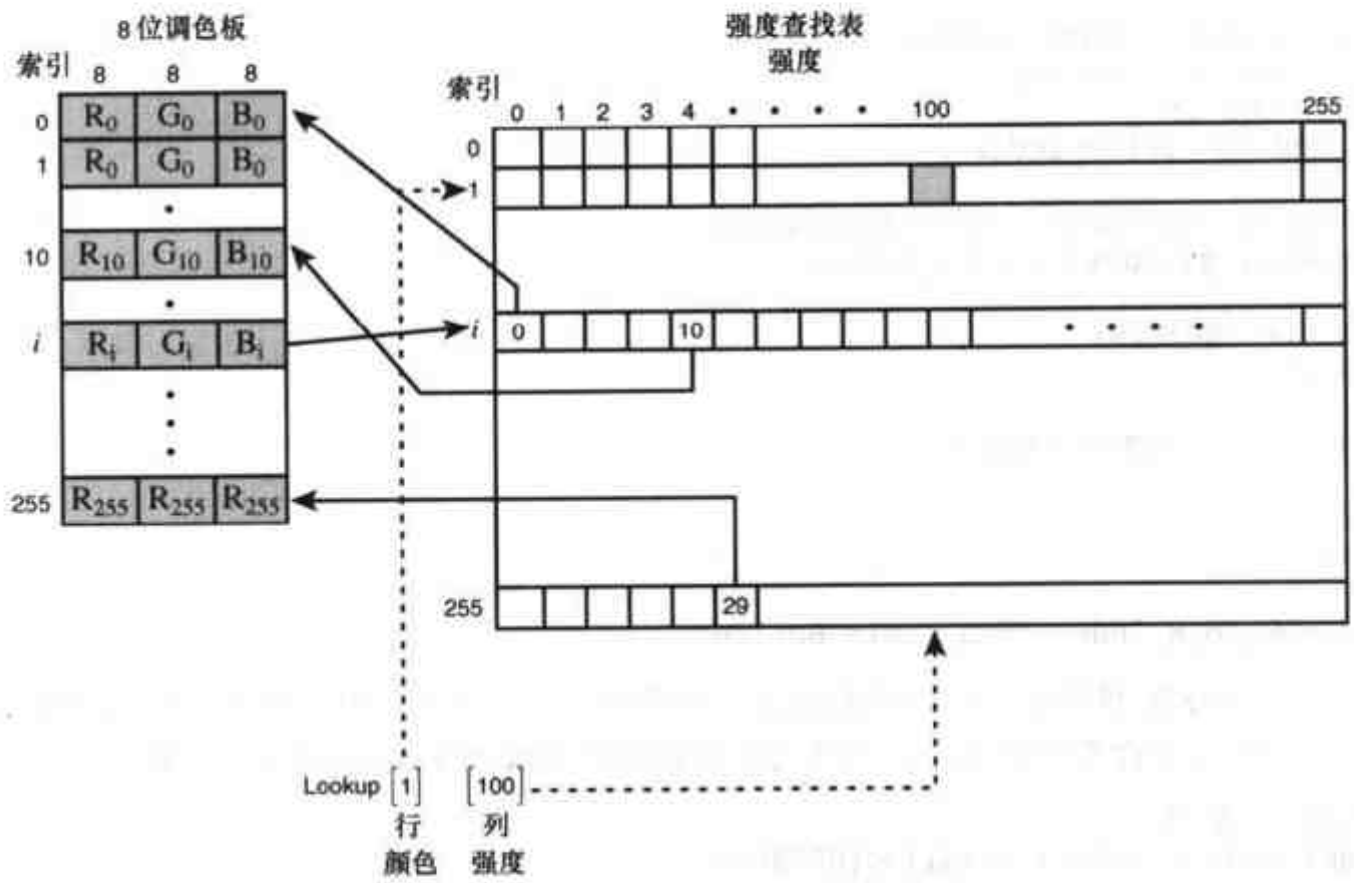


图 8.26　用强度作为索引的颜色转换查找表的结构

行，是因为在 8 位模式下，模型的颜色肯定是调色板中的颜色之一。当然，由于对于每种颜色，使用最小平方匹配的方式确定其 256 种着色（shade）度对应的颜色，因此着色（shade）度不多的颜色将变得很难看。

稍后将介绍创建这种查找表的函数，但需要指出的是，如果想同时使用这两种技术，需要编写两个不同的 8 位模式着色（shade），其中一个在 RBG 空间中执行所有计算，最后将结果转换为 16 位的 RGB 颜色，并通过 rgblookup[ ]将其转换为 8 位索引。然而，新模型将使用 2D 查找表，其中第一个索引为颜色，第二个索引为强度，即 rgbintensitylookup[color][intensity]。

要编写创建查找表的函数，只需遍历调色板中所有的颜色，对于每种颜色，计算其 256 种着色度对应的 RGB 值。问题是，计算颜色的 256 种着色度时，是否假定该颜色的强度已经是 100%，进而以此为起点向下降低亮度呢？

例如，假设有一种颜色，其 RGB 值为(100，0，0)，这是一种强度中等的红色。应该假定(100，0，0)是强度最高的红色，并据此创建 256 种着色度不同的红色？还是通过数学运算计算出虚拟颜色(255，0，0)，然后将其作为强度最高的红色，并据此创建 256 种着色度不同的红色？后一种方法更符合用户的意图，且给光照引擎提供的结果更佳。

这些确实是问题，但现在将采用这种方法：对于任何颜色索引，找出其对应的 RGB 值，然后提高该颜色的亮度，使其一个或多个分量达到最大允许的值。然后计算该颜色的 256 种不同亮度，并查找它们对应的调色板索引。这将会使模型变暗，但提供的一个颜色空间更为均匀。下面的例子使用源颜色 c，它计算强度最高的颜色 c^*。

假定：

多边形的颜色 c = RGB（red, green, blue） = RGB（50, 20, 100）

c^* = RGB（red^*, $green^*$, $blue^*$）

第 1 步　确定最大的分量：

最大分量 = blue = 100

第 2 步　将最大分量设置为 255，并计算放大比例：

blue = 100，$blue^*$ = 255，ratio = 255/100 = 2.55

第 3 步　按上述比例放大其他分量：

red^* = red * ratio = 50 * 2.55 = 127.5

$green^*$ = green * ratio = 20 * 2.55 = 51.0

blue = 255

四舍五入后，$c^* = RGB(128, 51, 155)$，其颜色与原来的颜色相同（即 R:G:B 相同），但强度更高。现在，将其强度视为 100%，并计算 256 种不同强度对应的 RGB 值。这类似于前面编写的函数，但不是查找通用 RGB 值对应的调色板索引，而是查找通过数学计算生成的 RGB 值对应的索引。这种算法的工作原理如下：对于每种颜色（经过强度调制后），假定其强度为 100%，并向下计算 256 种不同的强度值（最后的强度值为 0）及其对应的 RGB 值。对于每个 RGB 值，使用标准的最小平方法找出调色板中与之最接近的索引，并将其存储到查找表的相应行中。例如，如果颜色索引 25 对应的 RGB 为(255，0，0)，将在查找表的第 25 行存储下述索引：

$[RGB(0, 0, 0)_{i1}, RGB(1, 0, 0)_{i2}, RGB(2, 0, 0)_{i3}, \ldots, RGB(256, 0, 0)_{i256}]$

其中 $RGB(x, x, x)_i$ 是与 RGB(x，x，x) 最接近的颜色索引。

但愿读者不会觉得有关 8 位模式的介绍篇幅过于冗长。8 位模式很复杂，对软件引擎和非加速手持设备来说很有用，因此作者希望读者掌握在游戏中执行颜色转换时，对其输入域和输出范围进行控制的技巧。

下面是根据调色板创建强度表的函数。注意，位深是无关紧要的，因为所有计算都是在 RGB 8.8.8 空间中进行的，而计算结果为 8 位的调色板索引。因此，只需要提供调色板和表数据存储空间（64KB，总共 256 行，每行 256 个强度，每项 1 个字节）。

```
int RGB_16_8_Indexed_Intensity_Table_Builder(LPPALETTEENTRY src_palette,
UCHAR rgbilookup[256][256],  // 查找表
int intensity_normalization=1)
{
// 这个函数根据传入的调色板计算这样一个查找表
// 即行为调色板颜色，列为 0~255 的颜色强度，结果为单字节的调色板索引
// 调用该函数时，必须通过参数 rgbilookup 提供 64k 的存储空间
// 该函数不会申请存储空间
// 该函数遍历调色板中的颜色， palette and then
// 对于其中的每种颜色，在确保 RGB 分量不溢出的情况下最大限度地提高其强度
// 然后向下计算 256 种着色，使用最小平方法
// 找出调色板中与当前颜色和着色度最接近的索引，并将其存储到查找表中相应的位置
// 注意，如果参数 normalization 为 0，函数将不执行强度最大化步骤
int ri,gi,bi;        // 初始颜色
int rw,gw,bw;        // 用于存储临时颜色值
float ratio;         // 缩放比例
float dl,dr,db,dg;   // 256 种着色度之间的强度梯度

// 检查传入的参数
if (!src_palette || !rgbilookup)
   return(-1);

// 对于调色板中的每种颜色,计算最大可能强度
// 然后向下计算出 256 种着色度
for (int col_index = 0; col_index < 256; col_index++)
    {
// 根据调色板索引提取 R、G、B 值
ri = src_palette[col_index].peRed;
gi = src_palette[col_index].peGreen;
bi = src_palette[col_index].peBlue;

// 找出最大的分量，将其设置为 255
// 并根据放大比例相应地放大其他分量
if (intensity_normalization==1)
   {
   // 红色分量最大?
  if (ri >= gi && ri >= bi)
     {
   // 计算放大比例
   ratio = (float)255/(float)ri;

   // 将强度最大化
   ri = 255;
   gi = (int)((float)gi * ratio + 0.5);
```

```
    bi = (int)((float)bi * ratio + 0.5);

    } // end if
else // 绿色分量最大
if (gi >= ri && gi >= bi)
    {
    // 计算放大比例
    ratio = (float)255/(float)gi;

    // 将强度最大化
    gi = 255;
    ri = (int)((float)ri * ratio + 0.5);
    bi = (int)((float)bi * ratio + 0.5);

    } // end if
else // 蓝色分量最大
    {
    // 计算放大比例
    ratio = (float)255/(float)bi;

    // 将强度最大化
    bi = 255;
    ri = (int)((float)ri * ratio + 0.5);
    gi = (int)((float)gi * ratio + 0.5);
    } // end if

} // end if

// 计算强度梯度，以计算该颜色 256 种着色度对应的 RGB 值
dl = sqrt(ri*ri + gi*gi + bi*bi)/(float)256;
dr = ri/dl,
db = gi/dl,
dg = bi/dl;

// 初始化临时颜色值变量
rw = 0;
gw = 0;
bw = 0;

// 计算与当前颜色和着色度最接近的调色板索引
// 并将其存储到查找表中相应的位置
for (int intensity_index = 0; intensity_index < 256; intensity_index++)
    {
    int  curr_index  = -1;          // 当前最接近的颜色索引
    long curr_error  = INT_MAX;     // 当前距离

    for (int color_index = 0; color_index < 256; color_index++)
        {
        // 计算距离
        long delta_red   = abs(src_palette[color_index].peRed   - rw);
        long delta_green = abs(src_palette[color_index].peGreen - gw);
        long delta_blue  = abs(src_palette[color_index].peBlue  - bw);
        long error = (delta_red*delta_red) +
                     (delta_green*delta_green) +
                     (delta_blue*delta_blue);
        // 是否更近?
      if (error < curr_error)
          {
          curr_index = color_index;
          curr_error = error;
          } // end if

      } // end for color_index

// 找到最接近的颜色索引后，将其存储到查找表的相应位置中
rgbilookup[col_index][intensity_index] = curr_index;
```

```
// 计算下一种着色度对应的R、G、B值，并查找它们是否溢出
if (rw+=dr > 255) rw=255;
if (gw+=dg > 255) gw=255;
if (bw+=db > 255) bw=255;

} // end for intensity_index
} // end for c_index
// 成功返回
return(1);

} // end RGB_16_8_Indexed_Intensity_Table_Builder
```

要创建这种查找表，可以这样调用上述函数：

```
UCHAR rgbilookup[256][256];
RGB_16_8_Indexed_Intensity_Table_Builder(palette, rgbilookup, 1);
```

假设有一个颜色索引为 12 的多边形，您希望最终颜色为索引 12 和强度 150（最大值为 255）对应的颜色，则可以这样调用上述函数：

```
Final_pixel = rgbilookup[12][15];
```

就是这样简单！

注意：如果将这个函数的最后一个参数 intensity_normalization 设置为 0，将不执行强度最大化步骤。例如，对于颜色 RGB(100，10，10)，将假定其强度是最高的，并向下计算 256 种着色度；而不是将最大分量设置为最大运算值，并相应地放大其他分量，得到 RGB(255，25，255)，然后向下计算 256 种着色度。

8.3.5　固定着色

不知道读者怎样，反正作者已经厌倦了谈论着色，只想去实现它。接下来将创建 8 位和 16 位的着色实现，让读者能够在速度较慢的机器上使用前者，在奔腾 4 以上的机器上使用后者。首先来看固定着色（constant shading），这绝对是最简单的模型，不考虑任何光照效果——没有环境光、散射光等，只是使用某种颜色将多边形绘制为实心的。因此，固定着色根本不考虑光照模型，只是根据多边形颜色的索引或 RGB 值绘制它，这正是实现发射多边形的方式——使用其颜色绘制它。当然，正如前面指出的，发射多边形并不发射光，只是看似这样。要让多边形真正发射光，必须加上光源。

无论如何，我们已经为编写固定着色演示程序做好准备了。接下来将以第 7 章的坦克演示程序为基础，将其中绘制线框多边形的调用替换为绘制实心多边形的调用。原来的调用（16 位）如下：

```
// 绘制渲染列表
Draw_RENDERLIST4DV1_Wire16(&rend_list, back_buffer, back_lpitch);
```

新的调用如下：

```
//绘制渲染列表
Draw_RENDERLIST4DV1_Solid16(&rend_list, back_buffer, back_lpitch);
```

警告：这里必须为演示程序中的每种物体（坦克、高塔、地面标记）创建新的.PLG 文件，以确保每个多边形有正确的环绕顺序（winding order）和双面标记。这是因为我们要绘制实心物体，因此需要消除背面多边形，避免渲染它们。

更换上述调用后，演示程序如图 8.27 所示，该程序的源文件和可执行文件为 DEMOII8_4.CPP|EXE（DEMOII8_4_8b.CPP|EXE）。下面的#define 定义了 PLX/PLG 格式使用的一些标记：

```
// 双面标记
#define PLX_2SIDED_FLAG          0x1000   // 双面
#define PLX_1SIDED_FLAG          0x0000   // 单面

// 着色模式
#define PLX_SHADE_MODE_PURE_FLAG     0x0000  // 固定着色
```

```
#define PLX_SHADE_MODE_CONSTANT_FLAG 0x0000  // 别名
#define PLX_SHADE_MODE_FLAT_FLAG     0x2000  // 恒定着色
#define PLX_SHADE_MODE_GOURAUD_FLAG  0x4000  // Gouraud 着色
#define PLX_SHADE_MODE_PHONG_FLAG    0x6000  // Phong 着色
#define PLX_SHADE_MODE_FASTPHONG_FLAG0x6000  // Phong 着色（别名）
```

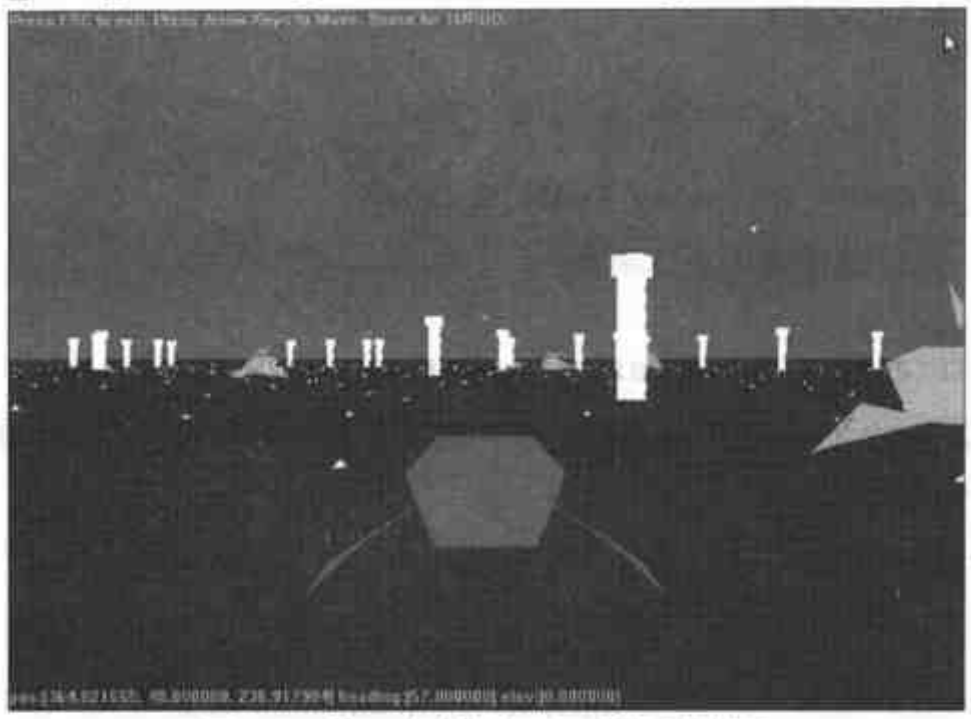


图 8.27　第一个着色演示程序的屏幕截图

当前，该函数 Draw_RENDERLIST4DV1*() 没有考虑着色模型，这是符合逻辑的，因为编写它们时还没有着色模型。PLG/PLX 格式支持指定着色方法，这里使用的是标记 PLX_SHADE_MODE_CONSTANT_FLAG，但函数没有考虑这个标记，因此它不起作用。在下一章编写代码时，将考虑着色方法，在光照模型和光照函数中考虑这些概念。本章介绍了很多新概念，由于篇幅有限，这里只打算使用已有的东西来实现它们，而不重新编写引擎；在下一章将对实现进行改进。

该演示程序存在一个重大缺陷：没有按正确的顺序绘制多边形。换句话说，我们需要实现某种深度排序，以便按从后到前的顺序渲染多边形。一种这样的方法是画家算法（painter's algorithm）。这种算法很容易实现，对于简单几何体，它能够对多边形进行排序，以正确的顺序渲染它们。这属于可见性问题，将在本章末尾编写这种算法的代码，为下一章介绍更复杂的可见性和深度缓存算法做准备。简单地说，其主要思想很简单：给定一个渲染列表，根据平均 z 值将多边形按从后到前的顺序排列，然后按这样的顺序渲染它们。这可以解决上述演示程序中的大部分可见性问题。图 8.28 说明了这种处理，这将在本章后面实现。

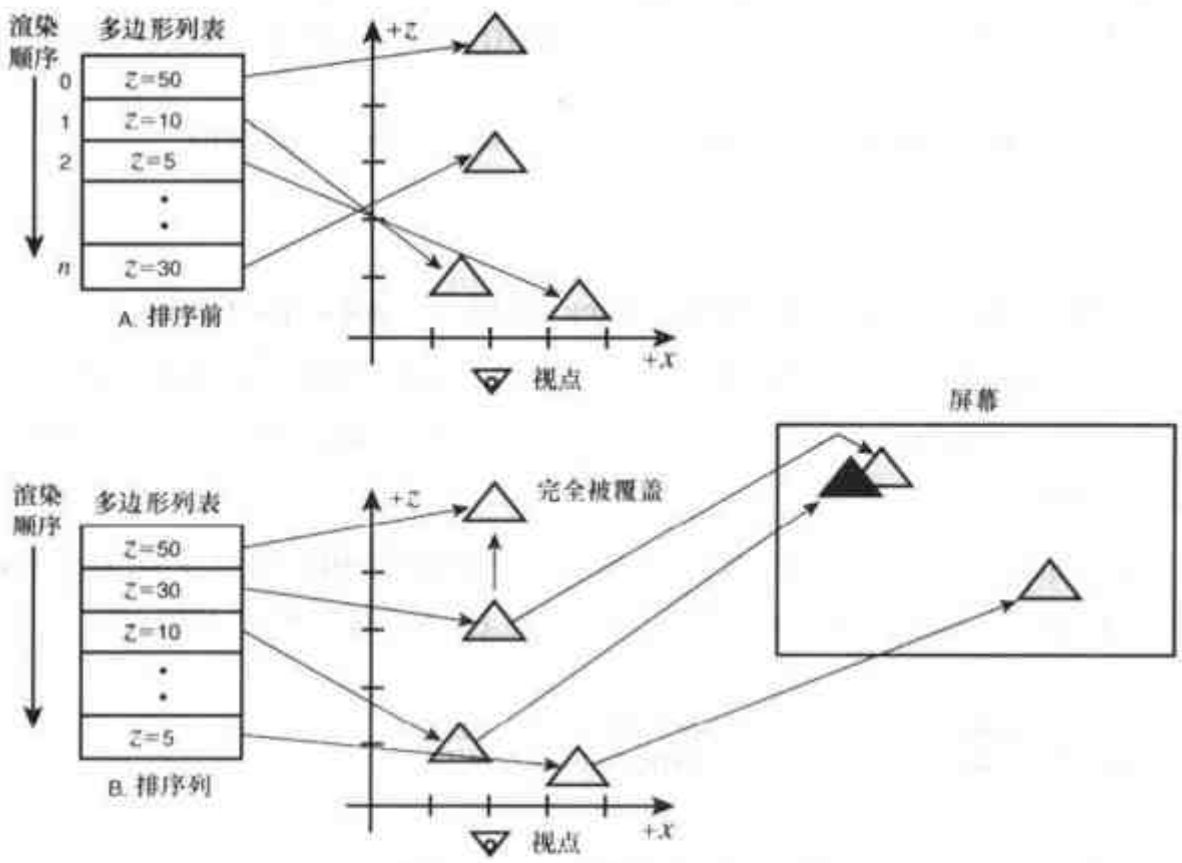


图 8.28　画家算法和 z 排序

8.3.6　恒定着色

恒定着色（flat shading）指的是根据多边形上某个像素的光照情况对整个多边形进行着色。换句话说，假设多边形是由一种材质构成的。我们的引擎只处理三角形，在每个多边形中，每个点的面法线都是相同的，因此对于每个多边形，只需对一个顶点执行光照计算，然后根据计算结果对整个多边形进行着色。这就是恒定着色，也叫面片着色（faceted shading）。对于由平面组成的物体，这种方法是可行的；但对于由曲面组成的物体，使用多边形对其进行近似时，这种方法导致物体看起来是由多边形组成的，如图 8.29 所示。如果使用 Gouraud 或 Phong 等平滑着色方法，平面多边形看起来将是平滑的，这将在本章后面介绍。现在介绍实现恒定着色的方法。

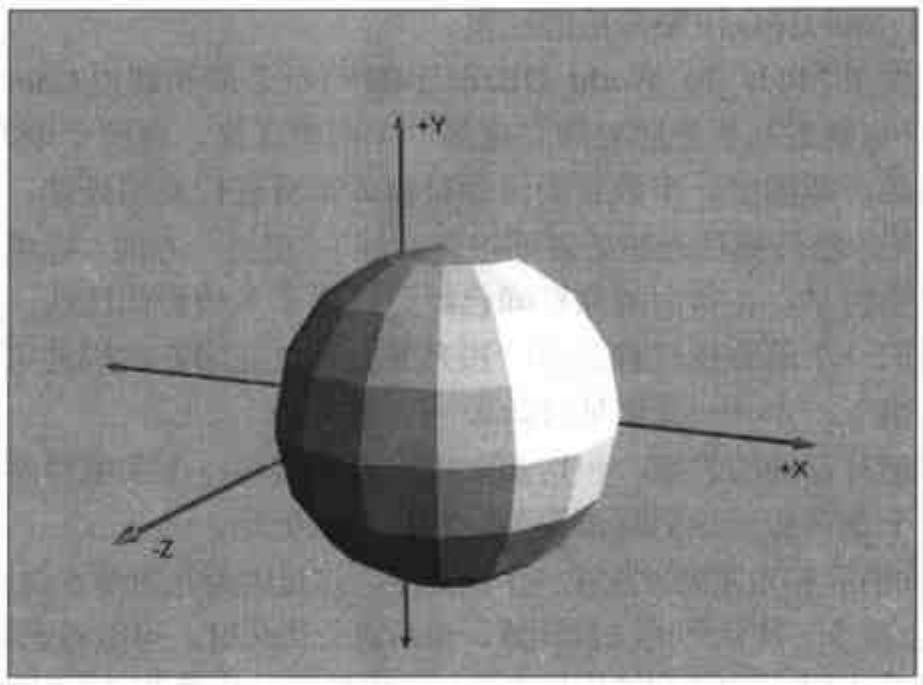


图 8.29　使用多边形来近似物体使得物体好像是由多边形组成的

我们有一个 PLG/PLX 文件加载函数，可以导入简单的模型，这种模型中指定了多边形的着色方法和颜色属性（RGB 值或颜色索引）。但我们还没有执行光照计算的函数，需要考虑如何编写这样的函数。另外，还需要决定在流水线的什么地方执行光照计算。正如前面指出的，光照计算可以在世界坐标系或相机坐标系下进行。还需要解决的问题是：应提供物体级的光照支持、渲染列表级的光照支持还是两者都提供？

需要解决的第一个问题是，在流水线的什么位置执行光照处理。下面是前一个演示程序的流水线，其中一些无关的代码被删除，以说明整体流程：

```
// 重置渲染列表
Reset_RENDERLIST4DV1(&rend_list);

// 创建相机变换矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);

// 重置物体
Reset_OBJECT4DV1(&obj);

// 执行世界坐标变换
Model_To_World_OBJECT4DV1(&obj, TRANSFORM_TRANS_ONLY);

// 执行物体剔除
Cull_OBJECT4DV1(&obj, &cam, CULL_OBJECT_XYZ_PLANES))

// 将物体插入到渲染列表中
Insert_OBJECT4DV1_RENDERLIST4DV1(&rend_list, &obj);

// 背面消除
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);

// 执行世界坐标到相机坐标变换
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);

// 执行相机坐标到透视坐标变换
```

```
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// 执行屏幕坐标变换
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);

// 渲染物体
Draw_RENDERLIST4DV1_Solid16(&rend_list, back_buffer, back_lpitch);
```

上述代码将一个物体插入到渲染列表中，同时还执行了很多其他的步骤。首先，引擎执行物体剔除测试，如果物体通过了这种测试，则将其插入到渲染列表中。将物体插入渲染列表中后，执行背面消除、世界坐标到相机坐标变换、相机坐标到透视坐标变换和透视坐标到屏幕坐标变换。最后，调用Draw_RENDERLIST4DV1_Solid16()渲染最后的结果。

从技术上说，可以在调用 Mode_To_World_OBJECT4DV1()之后和调用 Camera_To_Perspective_RENDERLIST4DV1()之前的任何地方执行光照处理。这是一个两难选择。有两个地方适合放置光照处理代码。可以将其作为物体处理函数，即创建一个独立的函数对物体本身进行光照处理，并在执行物体剔除操作后调用该函数（以免对不可见的物体执行光照处理而浪费时间）。在另一方面，也可以将光照处理代码放到背面消除函数中。这样做的原因是，在背面消除期间已经计算出了多边形的法线，而这些法线对于执行光照处理至关重要。然而，在同一个函数执行背面消除和光照处理可能降低代码的可读性。可以在背面消除期间保存计算得到的法线，供下一步——光照处理阶段——使用。

对于这种问题，没有绝对正确的答案，一切都取决于具体情况。下面编写两个光照函数：一个对物体本身执行光照处理，另一个在背面消除后对渲染列表执行光照处理。

现在暂时不考虑将背面消除和光照处理放在一个函数中，这是因为光照涉及的计算量非常大，只有需要进行严格的优化时才有必要去减少一次额外的法线计算。以后进行优化时，可能考虑将背面消除和光照处理放在一个函数中。就现在而言，我们将对物体和渲染列表执行光照处理，并将光照处理和背面消除分开。

注意：读者可能注意到了，作者总是倾向于对渲染列表而不是物体本身执行操作。原因在于，渲染列表中包含所有要处理的几何体，与对物体逐个进行处理相比，对渲染列表进行处理更为简单明了。

接下来需要考虑一些细节。例如，编写针对物体的光照模块时，需要修改每个多边形的颜色，有用于存储新颜色的空间吗？对于渲染列表，也存在这样的问题。最后，如果要在背面消除函数中执行光照处理，有用于存储多边形法线的空间吗？

首先来解决第一个问题。假设要对其进行光照处理的物体处于世界空间中，将多边形的最终颜色存储到哪里呢？不能覆盖多边形原来的颜色，因为这样做将丢失多边形的初始状态。来看一看数据结构 OBJECT4DV1：

```
// 基于顶点列表和多边形列表的物体
typedef struct OBJECT4DV1_TYP
{
int  id;              // 物体的数字 ID
char name[64];        // 物体的 ASCII 名称
int  state;           // 物体的状态
int  attr;            // 物体的属性
float avg_radius;     // 物体的平均半径，用于碰撞检测
float max_radius;     // 物体的最大半径
POINT4D world_pos;    // 物体在世界坐标系中的位置
VECTOR4D dir;         // 物体的旋转角度，为局部坐标或用户定义的单位方向向量

VECTOR4D ux,uy,uz;    // 记录物体朝向的局部坐标轴，物体旋转时将自动更新

int num_vertices;     // 顶点数
POINT4D vlist_local[OBJECT4DV1_MAX_VERTICES]; // 顶点的局部坐标
POINT4D vlist_trans[OBJECT4DV1_MAX_VERTICES]; // 变换后的顶点坐标
int num_polys;        // 多边形数
POLY4DV1 plist[OBJECT4DV1_MAX_POLYS];  // 多边形数组
} OBJECT4DV1, *OBJECT4DV1_PTR;
```

实际的多边形存储在 plist[ ]中，后者是一个由下述结构组成的数组：

```
// 基于顶点列表的多边形
typedef struct POLY4DV1_TYP
{
int state;      // 状态信息
int attr;       // 物理属性
int color;       // 多边形的颜色

POINT4D_PTR vlist; // 顶点列表
int vert[3];       // 指向顶点列表的索引
} POLY4DV1, *POLY4DV1_PTR;
```

不幸的是，上述结构中并没有用于存储光照处理后的颜色的空间。我们现在并不打算重新编写引擎。本章将使用现有的引擎和数据结构，下一章再创建新的引擎和数据结构，加入存储着色颜色的变量。因此，这里需要采取一些技巧。

颜色被存储为 32 位的整数，但该整数变量中只有后 16 位包含颜色信息，因为现在支持的颜色模型是 16/8 位的。因此可以将着色颜色存储到前 16 位中。然而，渲染函数使用后 16 位来渲染多边形，因此需要修改物体插入函数，使其假设多边形经过了光照处理，处理后的颜色存储在前 16 位中，因此将物体分解为多边形并将其插入渲染列表中时，将前 16 位作为多边形的颜色。新编写的物体插入函数如下：

```
int Insert_OBJECT4DV1_RENDERLIST4DV2(RENDERLIST4DV1_PTR rend_list,
                                     OBJECT4DV1_PTR obj,
                                     int insert_local=0,
                                     int lighting_on=0)
{
// 将物体转换为一个多边形面列表，
// 然后将可见、活动、没有被剔除和裁剪掉的多边形插入到渲染列表中
// 参数 insert_local 指定使用顶点列表 vlist_local 还是 vlist_trans
// 如果 insert_local 为 1，将把未变换的原始物体插入到渲染列表中
// 该参数的默认值为 0，即只插入至少经过了局部坐标到世界坐标变换的物体
// 最后一个参数指出之前是否执行了光照计算
// 光照计算生成的颜色值被存储在前 16 位中
// 如果 lighting_on = 1，则将多边形插入渲染列表时，
// 将使用这种颜色值覆盖多边形原来的颜色
// 该物体不处于活动状态、被剔除掉或不可见？
if (!(obj->state & OBJECT4DV1_STATE_ACTIVE) ||
    (obj->state & OBJECT4DV1_STATE_CULLED) ||
    !(obj->state & OBJECT4DV1_STATE_VISIBLE))
return(0);

// 提取物体包含的多边形
for (int poly = 0; poly < obj->num_polys; poly++)
     {
     // 获得多边形
    POLY4DV1_PTR curr_poly = &obj->plist[poly];
    // 多边形是否可见？
    if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
       (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
       (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
    continue; // 进入下一个多边形

// 如果要使用局部坐标，则改变多边形指向的顶点列表
// 首先保存原来的指针
POINT4D_PTR vlist_old = curr_poly->vlist;
if (insert_local)
   curr_poly->vlist = obj->vlist_local;
else
   curr_poly->vlist = obj->vlist_trans;

// 判断是否需要使用前 16 位中的颜色覆盖原来的颜色
if (lighting_on)
    {
    // 暂时保存颜色
    unsigned int base_color = (unsigned int)(curr_poly->color);
    curr_poly->color = (int)(base_color >> 16);
```

```
    } // end if

// 插入多边形
if (!Insert_POLY4DV1_RENDERLIST4DV1(rend_list, curr_poly))
    {
    // 恢复顶点列表指针
    curr_poly->vlist = vlist_old;

    // 插入失败
    return(0);
    } // end if

// 判断是否需要使用前 16 位中的颜色覆盖原来的颜色
if (lighting_on)
    {
    // 恢复颜色
    curr_poly->color = (int)base_color;
    } // end if

// 恢复顶点列表指针
curr_poly->vlist = vlist_old;

} // end for

// 成功返回
return(1);

} // end Insert_OBJECT4DV1_RENDERLIST4DV12
```

其中的粗体代码保存原来的颜色，然后将前 16 位右移到后 16 位中。这种方法对于 8/16 位模式都是可行的，因为我们并不关心后 16 位的值，只需将物体插入渲染列表中后恢复它们。如果之前执行了光照处理，则认为前 16 位是光照处理后的 RGB 值或颜色索引。无论是在 8 位模式还是 16 位模式下，只需将前 16 位右移到后 16 位中即可。

至此，我们提供了将经过光照处理后的物体插入到渲染列表中的功能（还没有编写光照函数本身），但还未处理另外两种光照处理方式：在背面消除期间和背面消除后对渲染列表执行光照处理。由于处理的是渲染列表，因此可以修改渲染列表中每个多边形的颜色，因为其初始颜色存储在物体中。这样，编写光照函数时，无需考虑颜色覆盖的问题。

前面探讨了在同一个函数中执行背面消除和光照计算的问题，但决定不这样做，以确保代码简单明了。这里的问题是，在独立地执行光照处理时是否可以利用背面消除期间计算出的法线？也就是说，在渲染列表中是否有用于存储法线的空间？图 8.30 说明了三种执行背面消除和光照计算的方式。下面来快速浏览一下 1.0 版的渲染列表数据结构：

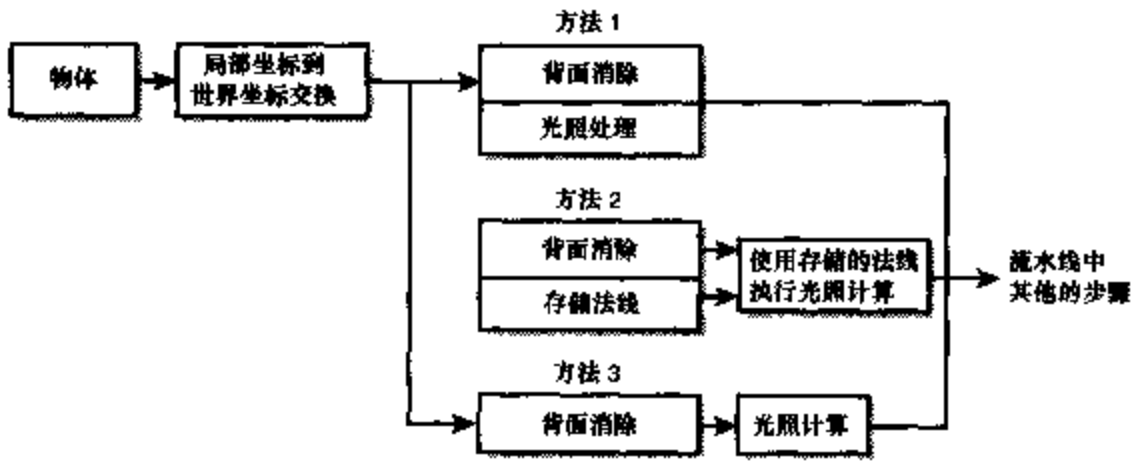


图 8.30　执行背面消除和光照计算的方式

```
typedef struct RENDERLIST4DV1_TYP
{
int state; // 渲染列表的状态
int attr;  // 渲染列表的属性
// 渲染列表是一个指针数组，其中的每个指针都指向
// 一个自包含的、可渲染的多边形面 POLYF4DV1
POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];
```

```
// 为避免每帧都分配和释放多边形存储空间
// 下面是实际存储多边形面的数组
POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS];

int num_polys; // 渲染列表中的多边形数目
} RENDERLIST4DV1, *RENDERLIST4DV1_PTR;
```

渲染列表由多边形面而不是简单多边形组成。多边形面结构如下：

```
typedef struct POLYF4DV1_TYP
{
int state;      // 状态信息
int attr;       // 多边形的物理属性
int color;      // 多边形颜色

POINT4D vlist[3];  // 三角形顶点
POINT4D tvlist[3]; // 变换后的顶点
POLYF4DV1_TYP *next; // 指向下一个多边形的指针
POLYF4DV1_TYP *prev; // 指向前一个多边形的指针
} POLYF4DV1, *POLYF4DV1_PTR;
```

每个多边形都提供了存储变换前和变换后顶点的空间，没有提供存储法线的空间。因此，虽然在背面消除函数中存储法线信息（而不是在同一个函数中执行背面消除和光照计算）是一个不错的注意，但并没有用于存储这些信息的空间，除非采取某种技巧将法线存储在指针 next 和 prev 中，但这样做太麻烦了。

这里想指出的是，原来的数据结构已经不能满足要求。

1. 16 位模式下的恒定着色

确定只编写用于物体和渲染列表的光照函数后，接下来需要决定支持哪种类型的光照。本章前面介绍了多种光源类型——从无穷远光源到聚光灯。另外，还介绍了如何创建光源和材质。我们需要编写 16 位和 8 位颜色模式下的代码，首先介绍 16 位的情况。这里只支持这样的光照模式：

- 采用固定着色的多边形不受光照模型的影响。
- 采用恒定着色的多边形将受光源列表中环境光源、无穷远光源、点光源和聚光灯的影响。

无论是对于物体还是渲染列表，光照处理方式都相同（对于渲染列表，无需保存原来的颜色，因此可以覆盖原来的颜色）。为不失普遍性，这里先进行笼统的介绍。

对于每个多边形，需要检查其属性。如果其 POLY4DV1_ATTR_SHADE_MODE_CONSTANT 标记被设置，则不对它执行光照计算；如果 POLY4DV1_ATTR_SHADE_MODE_FLAT 标记被设置，则对其执行光照计算。如图 8.31 所示，光照引擎 1.0 版的伪代码如下：

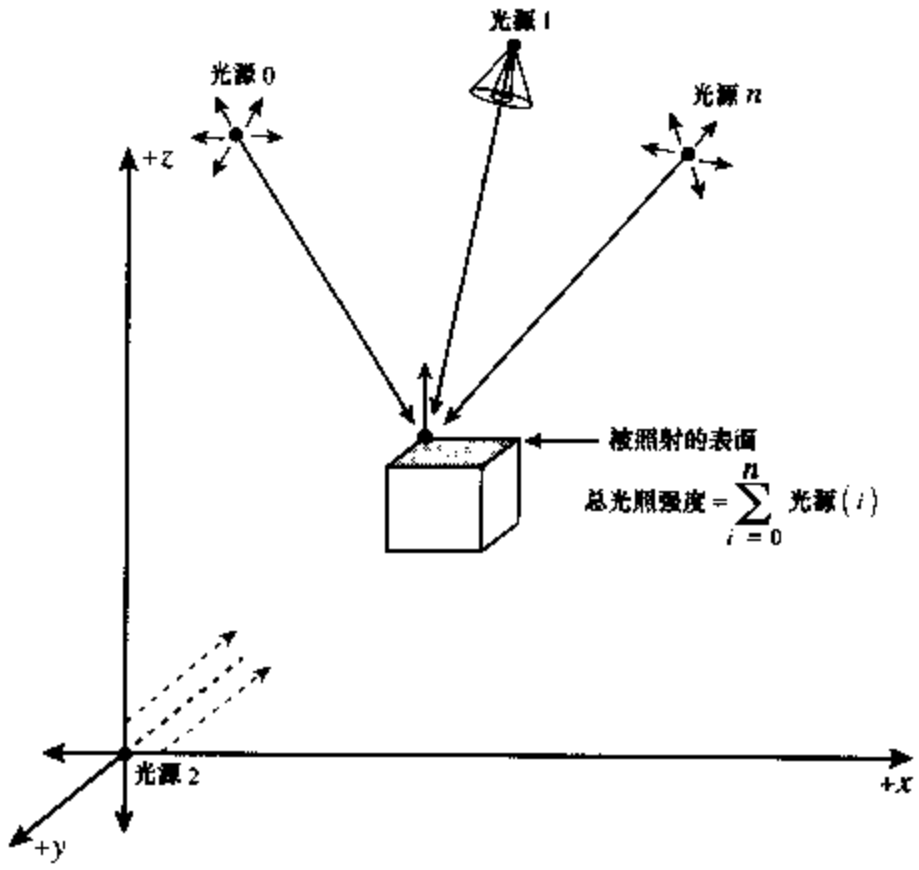


图 8.31　光照算法

第 1 步 计算面法线；

第 2 步 对于光源列表中的每个光源，计算其光照强度；

第 3 步 将所有光照强度相加；

第 4 步 将结果写入到多边形颜色变量的前 16 位中。

当然，必须根据本章前面介绍的光源模型来执行光照计算。另外，光照引擎 1.0 版不支持材质的概念，对于每个物体都使用默认材质，因此颜色本身就是反射系数。最后，物体只反射环境光和散射光。

下面是根据传入的光源列表对物体执行光照计算的函数：

```
int Light_OBJECT4DV1_World16(OBJECT4DV1_PTR obj,  // 要处理的物体
CAM4DV1_PTR cam,       // 相机位置
LIGHTV1_PTR lights,    // 光源列表（可能有多个）
int max_lights)        // 最大光源数
{
// 16 位光照函数
// 该函数根据传入的光源列表和相机对物体执行光照计算
// 它支持固定着色以及根据环境光源、无穷远光源、点光源和聚光灯进行恒定着色
// 这个函数只是根据光照公式进行计算
// 但使用的是整数，而不是浮点数
// 为何这样做呢？整数和浮点数的运算速度相同，
// 但执行浮点数转换可能消耗大量的 CPU 周期，
// 因此如果计算只涉及整数，将可提高速度
// 另外，1 类聚光灯只不过是一种有方向的点光源，
// 但强度衰减效果使其仍像一个聚光灯
// 2 类聚光灯是这样实现的：
// 根据表面位置和光源方向的相对角度的余弦计算强度
// 这类似于简化后的聚光灯模型，但 pf 必须是整数

unsigned int r_base, g_base, b_base,   // 原来的颜色值
             r_sum,  g_sum,  b_sum,    //  全部光源的总体光照效果
             shaded_color;             // 最后的颜色

float dp,     // 点积
      dist,   // 表面和光源之间的距离
      i,      // 强度
      nl,     // 法线长度
      atten;  // 衰减计算结果

// 检查物体是否被剔除
if (!(obj->state & OBJECT4DV1_STATE_ACTIVE) ||
   (obj->state & OBJECT4DV1_STATE_CULLED) ||
   !(obj->state & OBJECT4DV1_STATE_VISIBLE))
   return(0);

// 处理每个多边形
for (int poly=0; poly < obj->num_polys; poly++)
    {
    // 获取多边形
   POLY4DV1_PTR curr_poly = &obj->plist[poly];

   // 多边形是否有效？
   // 多边形没有被裁剪和剔除掉、处于活动状态、不是背面且可见？
   if (!(curr_poly->state & POLY4DV1_STATE_ACTIVE) ||
      (curr_poly->state & POLY4DV1_STATE_CLIPPED ) ||
      (curr_poly->state & POLY4DV1_STATE_BACKFACE) )
      continue; // 进入下一个多边形

   // 提取指向主列表的顶点索引
   // 多边形不是自包含的，而是基于物体的顶点列表的
   int vindex_0 = curr_poly->vert[0];
   int vindex_1 = curr_poly->vert[1];
   int vindex_2 = curr_poly->vert[2];

   // 使用变换后的多边形顶点列表
   // 因为背面消除只有在进入流水线的世界坐标阶段后才有意义
```

```
// 检查多边形的着色模式
if (curr_poly->attr & POLY4DV1_ATTR_SHADE_MODE_FLAT ||
   curr_poly->attr & POLY4DV1_ATTR_SHADE_MODE_GOURAUD)
   {
   // 第 1 步：提取多边形颜色的 RGB 值
  if (dd_pixel_format == DD_PIXEL_FORMAT565)
     {
     _RGB565FROM16BIT(curr_poly->color, &r_base, &g_base, &b_base);

     // 转换为 8.8.8 格式
     r_base <<= 3;
     g_base <<= 2;
     b_base <<= 3;
     } // end if
else
    {
    RGB555FROM16BIT(curr_poly->color, &r_base, &g_base, &b_base);

    // 转换为 8.8.8 格式
    r_base <<= 3;
    g_base <<= 3;
    b_base <<= 3;
    } // end if

// 初始化总体光照颜色
r_sum  = 0;
g_sum  = 0;
b_sum  = 0;

// 遍历光源
for (int curr_light = 0; curr_light < max_lights; curr_light++)
   {
   // 光源是否被打开
  if (lights[curr_light].state)
     continue;
 // 判断光源类型
if (lights[curr_light].attr & LIGHTV1_ATTR_AMBIENT)
   {
   // 将每个分量与多边形颜色相乘，并除以 256，以确保结果为 0-255 之间
   r_sum+= ((lights[curr_light].c_ambient.r * r_base) / 256);
   g_sum+= ((lights[curr_light].c_ambient.g * g_base) / 256);
   b_sum+= ((lights[curr_light].c_ambient.b * b_base) / 256);
   // 最好只设置一个环境光源
   } // end if
else if (lights[curr_light].attr & LIGHTV1_ATTR_INFINITE)
     {
     // 无穷远光源，因此需要知道面法线和光源方向

     // 需要计算多边形面的法线
     // 顶点按顺时针方向排列的，因此 u=p0->p1, v=p0->p2, n=uxv
     VECTOR4D u, v, n;

     // 计算 u 和 v
     VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                    &obj->vlist_trans[ vindex_1 ], &u);
     VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                    &obj->vlist_trans[ vindex_2 ], &v);

     // 计算叉积
     VECTOR4D_Cross(&u, &v, &n);

     // 至此，准备工作基本完成，但还需要对法线向量进行归一化
     // 这里存在很大的优化空间
     //为优化这个步骤，可预先计算出所有多边形法线的长度
     nl = VECTOR4D_Length_Fast(&n);

     // 无穷远光源的光照模型如下：
     // I(d)dir = I0dir * Cldir
```

```
        // 散射项的计算公式如下：
        // Itotald =   Rsdiffuse*Idiffuse * (n . l)
        // 因此只需要将它们乘起来即可
        // 这里先乘以 128，以避免浮点数计算
        // 这样做不是因为浮点数计算速度更慢，而是浮点数转换可能消耗大量 CPU 周期
        dp = VECTOR4D_Dot(&n, &lights[curr_light].dir);

        // 仅当 dp > 0 时，才需要考虑该光源的影响
        if (dp > 0)
           {
           i = 128*dp/nl;
           r_sum+= (lights[curr_light].c_diffuse.r * r_base * i) /
                   (256*128);
           g_sum+= (lights[curr_light].c_diffuse.g * g_base * i) /
                   (256*128);
           b_sum+= (lights[curr_light].c_diffuse.b * b_base * i) /
                   (256*128);
           } // end if

        } // end if infinite light
     else if (lights[curr_light].attr & LIGHTV1_ATTR_POINT)
        {
        // 执行点光源光照计算
        // 点光源的光照模型如下：
        //                  I0point * Clpoint
        //  I(d)point = ------------------------
        //                  kc + kl*d + kq*d2
        //
        //  其中 d = |p - s|
        // 除了强度随距离衰减外，几乎与无穷远光源相同
        // 需要计算多边形的法线
        // 顶点按顺时针方向排列，因此 u=p0->p1, v=p0->p2, n=uxv
        VECTOR4D u, v, n, l;

        // 计算 u 和 v
        VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                       &obj->vlist_trans[ vindex_1 ], &u);
        VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                       &obj->vlist_trans[ vindex_2 ], &v);

        // 计算叉积
        VECTOR4D_Cross(&u, &v, &n);

        // 至此，准备工作基本完成，但还需要对法线向量进行归一化
        // 这里存在很大的优化空间
        // 为优化这个步骤，可预先计算出所有多边形法线的长度
        nl = VECTOR4D_Length_Fast(&n);

        // 计算从表面到光源的向量
        VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                       &lights[curr_light].pos, &l);

        // 计算距离和衰减
        dist = VECTOR4D_Length_Fast(&l);

        // 对于散射光
        // Itotald =   Rsdiffuse*Idiffuse * (n . l)
        // 因此只需要将它们乘起来即可
        // 这里先乘以 128，以避免浮点数计算
        // 这样做不是因为浮点数计算速度更慢，而是浮点数转换可能消耗大量 CPU 周期
        dp = VECTOR4D_Dot(&n, &l);

        //仅当 dp > 0 时，才需要考虑该光源的影响
        if (dp > 0)
           {
           atten = (lights[curr_light].kc + lights[curr_light].kl*dist +
                    lights[curr_light].kq*dist*dist);
```

```
        i = 128*dp / (nl * dist * atten );

        r_sum += (lights[curr_light].c_diffuse.r * r_base * i) /
                 (256*128);
        g_sum += (lights[curr_light].c_diffuse.g * g_base * i) /
                 (256*128);
        b_sum += (lights[curr_light].c_diffuse.b * b_base * i) /
                 (256*128);
         } // end if
      } // end if point
   else
   if (lights[curr_light].attr & LIGHTV1_ATTR_SPOTLIGHT1)
      {
      // 执行聚光灯光照计算，这里使用简化的模型：
      // 即使用带方向的点光源来模拟聚光灯，因此
      //                  I0point * Clpoint
      //  I(d)point = ------------------------
      //                kc +  kl*d + kq*d2
      //
      //  其中 d = |p - s|
      // 除了强度随距离衰减外，几乎与无穷远光源相同
      // 需要计算多边形的法线

      // 顶点按顺时针方向排列，因此 u=p0->p1, v=p0->p2, n=uxv
      VECTOR4D u, v, n, l;

      // 计算 u 和 v
      VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                     &obj->vlist_trans[ vindex_1 ], &u);
      VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                     &obj->vlist_trans[ vindex_2 ], &v);

      // 计算叉积
      VECTOR4D_Cross(&v, &u, &n);

      // 至此，准备工作基本完成，但还需要对法线向量进行归一化
      // 这里存在很大的优化空间
      // 为优化这个步骤，可预先计算出所有多边形法线的长度
      nl = VECTOR4D_Length_Fast(&n);

      // 计算从表面到光源的向量
      VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                     &lights[curr_light].pos, &l);

      // 计算距离和衰减
      dist = VECTOR4D_Length_Fast(&l);

      // 对于散射光
      // Itotald =   Rsdiffuse*Idiffuse * (n . l)
      // 因此只需要将它们乘起来即可
      // 这里先乘以 128，以避免浮点数计算
      // 这样做不是因为浮点数计算速度更慢，而是浮点数转换可能消耗大量 CPU 周期

      // 注意，这里使用的是光源方向，而不是到光源的向量
      // 从而类似于聚光灯模型那样考虑了朝向
      dp = VECTOR4D_Dot(&n, &lights[curr_light].dir);

      //仅当 dp > 0 时，才需要考虑该光源的影响
      if (dp > 0)
         {
         atten =  (lights[curr_light].kc +
                   lights[curr_light].kl*dist +
                   lights[curr_light].kq*dist*dist);

         i = 128*dp / (nl * atten );

         r_sum += (lights[curr_light].c_diffuse.r * r_base * i) /
```

```
                    (256*128);
          g_sum += (lights[curr_light].c_diffuse.g * g_base * i) /
                    (256*128);
          b_sum += (lights[curr_light].c_diffuse.b * b_base * i) /
                    (256*128);
          } // end if
       } // end if spotlight1
    else
    if (lights[curr_light].attr & LIGHTV1_ATTR_SPOTLIGHT2)
       // 简化的聚光灯模型
       {
       // 执行聚光灯光照计算
       // 简化的聚光灯模型如下:
       //                IOspotlight * Clspotlight * MAX( (l . s), 0)^pf
       // I(d)spotlight = ------------------------------------------------
       //                              kc + kl*d + kq*d2
       // 其中 d = |p - s|, pf = 指数因子
       // 除了分子中包含与光源和照射点相对角度相关的项外
       // 这几乎与点光源模型相同

       // 需要计算多边形的法线
       // 顶点按顺时针方向排列，因此 u=p0->p1, v=p0->p2, n=uxv
       VECTOR4D u, v, n, d, s;

       // 计算 u 和 v
       VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                      &obj->vlist_trans[ vindex_1 ], &u);
       VECTOR4D_Build(&obj->vlist_trans[ vindex_0 ],
                      &obj->vlist_trans[ vindex_2 ], &v);

       // 计算叉积
       VECTOR4D_Cross(&v, &u, &n);
       // 至此，准备工作基本完成，但还需要对法线向量进行归一化
       // 这里存在很大的优化空间
       // 为优化这个步骤，可预先计算出所有多边形法线的长度
       nl = VECTOR4D_Length_Fast(&n);

       // 对于散射光
       // Itotald =   Rsdiffuse*Idiffuse * (n . l)
       // 因此只需要将它们乘起来即可
       // 这里先乘以 128，以避免浮点数计算
       // 这样做不是因为浮点数计算速度更慢，而是浮点数转换可能消耗大量 CPU 周期
       dp = VECTOR4D_Dot(&n, &lights[curr_light].dir);

       //仅当 dp > 0 时，才需要考虑该光源的影响
       if (dp > 0)
          {
          // 计算从表面到光源的向量
          VECTOR4D_Build( &lights[curr_light].pos,
                          &obj->vlist_trans[ vindex_0 ], &s);

          //计算 s 的长度（到光源的距离），以归一化
          dist = VECTOR4D_Length_Fast(&s);

          // 计算点积项(s . l)
          float dpsl = VECTOR4D_Dot(&s, &lights[curr_light].dir)/dist;

          // 仅当这项为正才进行下述处理
          if (dpsl > 0)
             {
             // 计算衰减
             atten = (lights[curr_light].kc +
                      lights[curr_light].kl*dist +
                      lights[curr_light].kq*dist*dist);

             // 为提高速度，pf 必须是大于 1.0 的整数
             float dpsl_exp = dpsl;
```

```
            // 计算整数次方
            for (int e_index = 1;
                 e_index < (int)lights[curr_light].pf; e_index++)
                 dpsl_exp*=dpsl;

            // 现在 dpsl_exp 存储的是(dpsl)^pf，即(s . l)^pf

            i = 128*dp * dpsl_exp / (nl * atten );

            r_sum += (lights[curr_light].c_diffuse.r * r_base * i) /
                      (256*128);
            g_sum += (lights[curr_light].c_diffuse.g * g_base * i) /
                      (256*128);
            b_sum += (lights[curr_light].c_diffuse.b * b_base * i) /
                      (256*128);
            } // end if
          } // end if
        } // end if spot light
      } // end for light

    // 确保颜色分量不溢出
    if (r_sum  > 255) r_sum = 255;
    if (g_sum  > 255) g_sum = 255;
    if (b_sum  > 255) b_sum = 255;

    // 写入颜色
    shaded_color = RGB16Bit(r_sum, g_sum, b_sum);
    curr_poly->color = (int)((shaded_color << 16) | curr_poly->color);

    } // end if
else // POLY4DV1_ATTR_SHADE_MODE_CONSTANT
     {
     // 采用固定着色，将原来的颜色复制到前 16 位中
     curr_poly->color = (int)((curr_poly->color << 16) | curr_poly->color);
     } // end if

  } // end for poly

// 成功返回
return(1);

} // end Light_OBJECT4DV1_World16
```

这个光照函数以简单的方式实现了本章开头讨论的内容。它首先根据像素格式提取多边形颜色的 RGB 分量，并将每个分量转换为 8 位格式。然后进入光照计算循环。如果多边形采用固定着色，则跳过它并进入下一个多边形；否则根据光源列表对多边形执行光照计算。这个算法将光源列表中每个处于活动状态的光源的贡献相加，它支持环境光源、点光源、无穷远光源，甚至支持聚光灯，但对其进行了高度优化/简化。在光源数据结构中，有多个针对聚光灯的常量，用于支持简化的聚光灯模型。这个光照系统支持两种聚光灯模型：

- LIGHTV1_ATTR_SPOTLIGHT1——这种聚光灯类似于点光源，只是使用了方向向量。因此可以像点光源那样设置它，但需要使用一个方向向量来指定聚光灯的方向。有趣的是，这种衰减效果很好地模拟了锥体衰减。
- LIGHTV1_ATTR_SPOTLIGHT2——这种聚光灯完整地没有本影和半影的聚光灯模型。本章前面介绍了两种聚光灯模型，其中一种将指数因子作为点积项的指数，这实现了聚光灯光照强度减弱效果。设置这种聚光灯时，只需指定指数因子 *pf*，而不指定本影角和半影角。另外，*pf* 必须是大于 1.0 的整数。在上述函数中，使用乘法来计算幂，因为 exp() 的开销非常高。

将光源数据库中所有光源的贡献相加，然后根据像素格式将最终的 RGB 值转换为 5.6.5 或 5.5.5 格式，并将其复制到多边形颜色字段的前 16 位中，以免覆盖多边形原来的颜色。

警告：执行光照计算时，RGB 分量可能会溢出，因此一定要检查溢出情况，避免分量超过 255 或最大值。当然使用内联函数来完成这种检测。

调用这个函数很简单，只需指定物体、相机（现在未使用，但以后可能需要）、光源列表和光源列表中的光源数即可。下面的调用假设光源存储在全局数组 lights 中，其中包含 3 个光源：

```
// 执行光照计算
Light_OBJECT4DV1_World(&obj_marker, &cam, lights, 3);
```

执行上述调用后，将对物体执行光照计算，并将每个多边形经过光照处理后的颜色存储到颜色字段的前 16 位中。这样，新的物体插入函数 Insert_OBJCET4DV1_RENDERLIST4DV2()将能够正确运行。

对渲染列表执行光照计算的函数的逻辑与此相同，其原型如下：

```
int Light_RENDERLIST4DV1_World16(
          RENDERLIST4DV1_PTR rend_list,  // 要处理的渲染列表
          CAM4DV1_PTR cam,      // 相机
          LIGHTV1_PTR lights, // 光源列表（可能有多个）
          int max_lights);      // 最大光源数
```

该函数的调用方式与物体光照函数相同，但必须在执行背面消除后且渲染列表为世界坐标或相机坐标时调用它。

下面来看一个光照演示程序。图 8.32 是演示程序 DEMOII8_5.CPP|EXE 的屏幕截图。这个演示程序创建了一个环境光源、点光源、无穷远光源和 1 类聚光灯，并将它们放置到游戏世界中。该演示程序启动时，所有光源都将被启用，用户将看到这些光源的照射效果。另外，坦克的垂直稳定器（vertical stabilizer）有发射光（即使用固定着色），因此看起来像一个光源，但不影响其他物体。有关如何控制该演示程序，请参阅表 8.2。

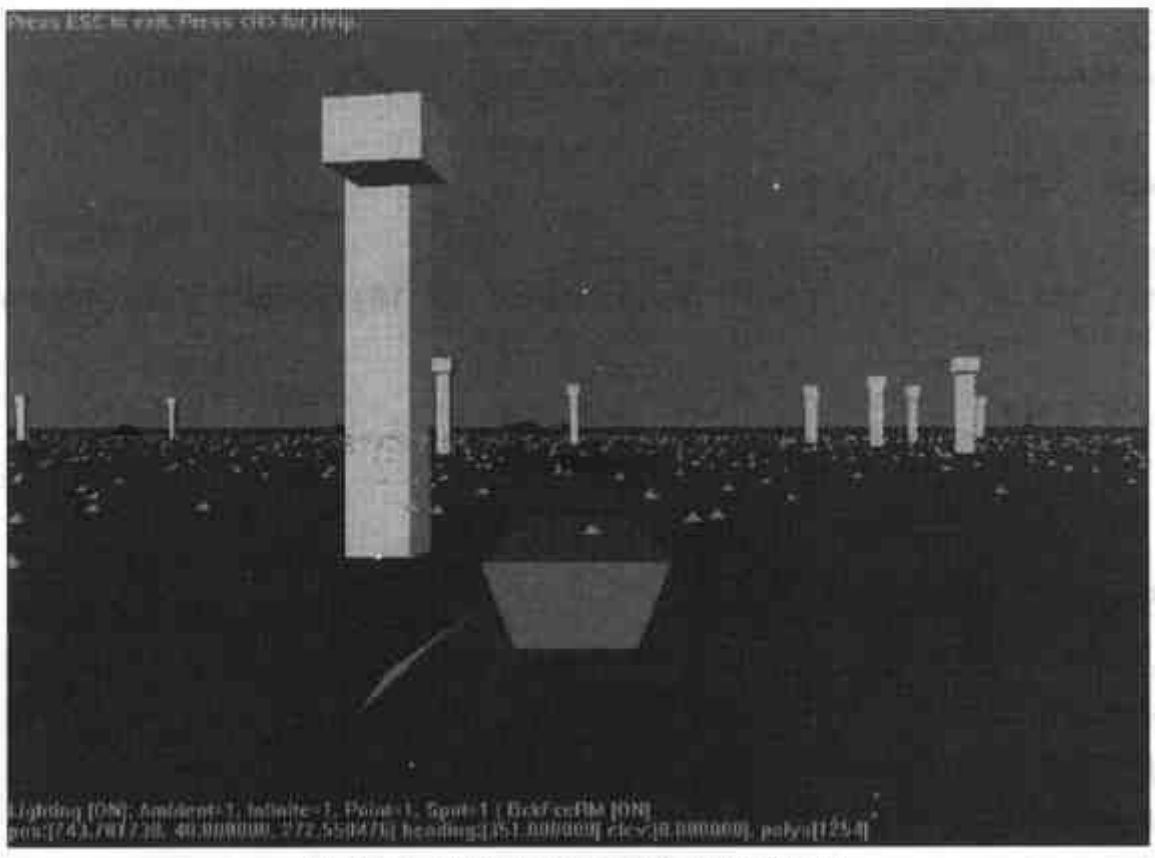


图 8.32　光照引擎演示程序的屏幕截图

表 8.2　　如何控制演示程序

键	功能	键	功能
A	开/关环境光源	右箭头键	让玩家向右转
I	开/关无穷远光源	左箭头键	让玩家向左转
P	开/关点光源	上箭头键	玩家前进
S	开/关聚光灯	上箭头键	玩家后退
W	在线框模式和实心模式之间切换	空格键	加速
B	开/关背面消除	H	打开 Help 菜单
L	开/关光照引擎	Esc	退出程序

提示：要编译该演示程序，应包含文件 T3DLIB.CPP|H ~ T3DLIB6.CPP|H，当然还有 DEMOII8_5.CPP。

作者喜欢光源在游戏世界中不断移动，尤其是点光源和聚光灯，它们是彩色的。请尝试修改光源的设置，如颜色等。这是一个软件演示程序，它在 16 位模式下实时地实现彩色光源，很不错。

2. 8 位模式下的恒定着色

在 8 位模式下实现着色是一种挑战，其原因很多。首先，只能用 8 位来存储颜色，但主要原因在于，只有一个调色板。无论采用何种方式来对表面执行光照计算，最终的颜色都是 8 位的，且来自最初的调色板。在 8 位模式下实现 shader 的方式有两种，可以采用通用方法（作者就是这样做的），也可以针对 8 位模式增加约束条件并进行简化。例如，限制光源为白色的，从而将光源简化为只有强度而没有颜色，等等。本章前面创建了两个颜色查找表（参见图 8.24～8.26），完全可以使用第二个查找表，其中每行是一种颜色的 256 种不同的着色度，可以使用下述方式来查找颜色和着色度对应的索引：

```
shaded_color = rgbilookup[color][shade];
```

当然，颜色和着色度的取值范围为 0～255，查找结果为调色板中最接近的索引。

然而，作者现在还不想重写光照系统。作者认为 16 位光照的速度足够快，暂时不想针对 8 位光照进行优化，我们将支持这种方式，但不采用本征（natively）方式进行支持。好消息是，编写 8 位光照模块只需在 16 位光照模块的基础上添加和删除一些代码。进攻计划如下：不再需要考虑像素模式；只考虑多边形的颜色索引。根据颜色索引，从系统调色板中找出其对应的 RGB 值，然后使用这些值（每个分量 8 位）来执行光照计算。执行完光照计算后，将最终的 RGB 值转换为 5.6.5 值，然后使用它在 RGB_16_8_IndexedRGB_Table_Builder()创建的查找表中查找相应的颜色索引。整个过程如图 8.33 所示。

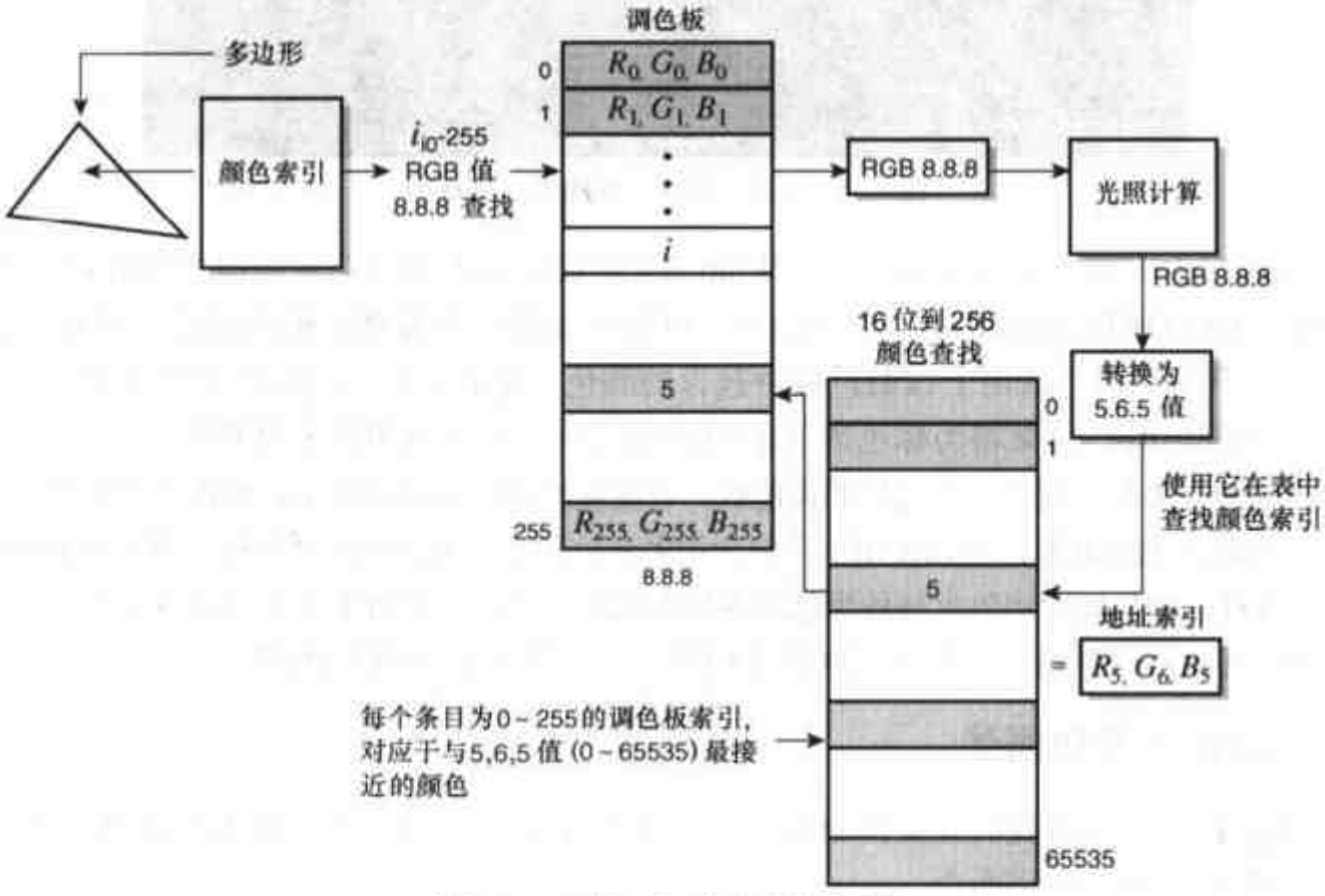


图 8.33　8 位 RGB 光照处理过程

由于篇幅有限，同时该函数与前一个函数基本相同，这里不打算列出其完整的代码。这个函数的原型如下：

```
int Light_OBJECT4DV1_World(OBJECT4DV1_PTR obj,  // 要处理的物体
    CAM4DV1_PTR cam,     // 相机
    LIGHTV1_PTR lights,  // 光源列表（可能有多个）
    int max_lights);     // 最大光源数
```

注意：对于 8/16 位函数，本书采用这样的命名约定，即 16 位的函数版本末尾包含“16”。这两种版本的函数的功能相同，代码类似，但名称有所不同。当然，可以将这两种函数合而为一，但将它们分开旨在减少函数中的代码，并提高函数的速度。

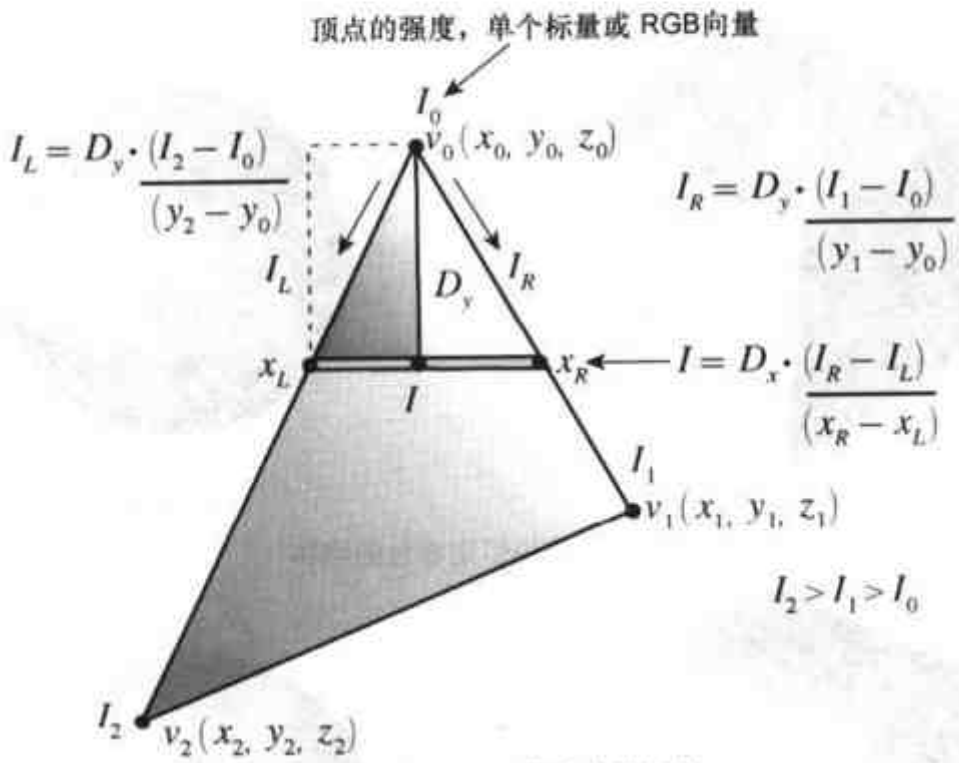


图 8.38　Gouraud 着色中的插值

读者对此应非常熟悉，它是一种标准插值算法，被用于从颜色到纹理图等各种东西的插值。本书将大量使用这种插值算法。下一章讨论仿射几何学（affine）纹理映射时，将深入探讨这种插值算法，如果读者现在还不太明白，到那时将会明白。

8.3.8　Phong 着色概述

Gouraud 着色是一种事实标准，市面上 99%的图形加速器都使用这种方法。它看起来很不错，但也存在没有考虑透视的缺点。Phong 着色解决了这种问题，它对每个像素执行着色计算。它不是在屏幕空间中通过插值来计算多边形中每个点的颜色，而是在屏幕空间中通过插值来计算多边形中每个点的法线，然后根据这些法线来执行光照计算，如图 8.39 所示。Phong 着色的真实感比 Gouraud 着色强些。它提高了镜面反射效果，使用 Gouraud 着色无法得到明显的镜面反射区域，因为它只是沿扫描线从一个值增加（降低）到另一个值，而 Phong 着色是针对像素的，每个像素都可能发生变化。

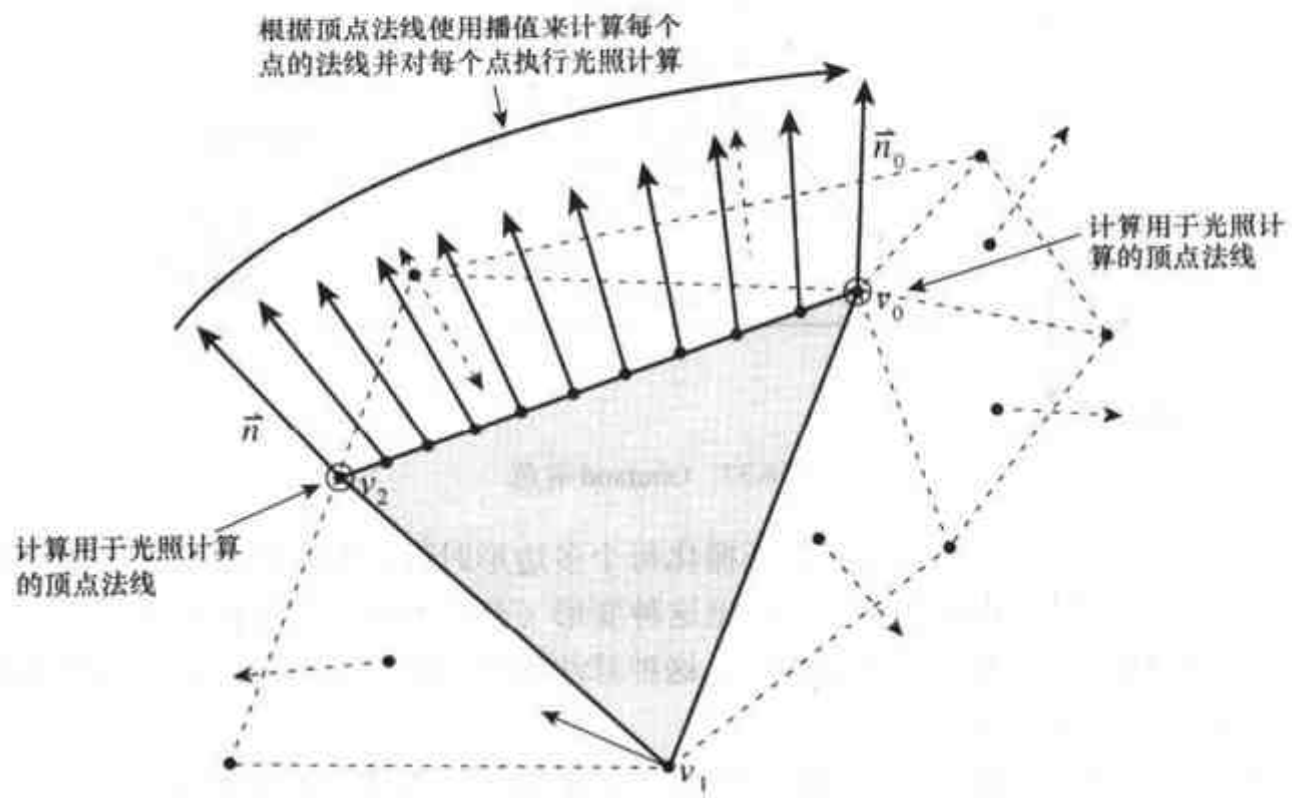


图 8.39　Phong 顶点法线插值算法

Phong 着色的缺点是，使用硬件都可能难以实现，更不用说使用软件了，因此本书不实现它。但在介绍高级光照处理时，将使用一些技巧来实现类似于 Phong 着色的镜面反射区域。

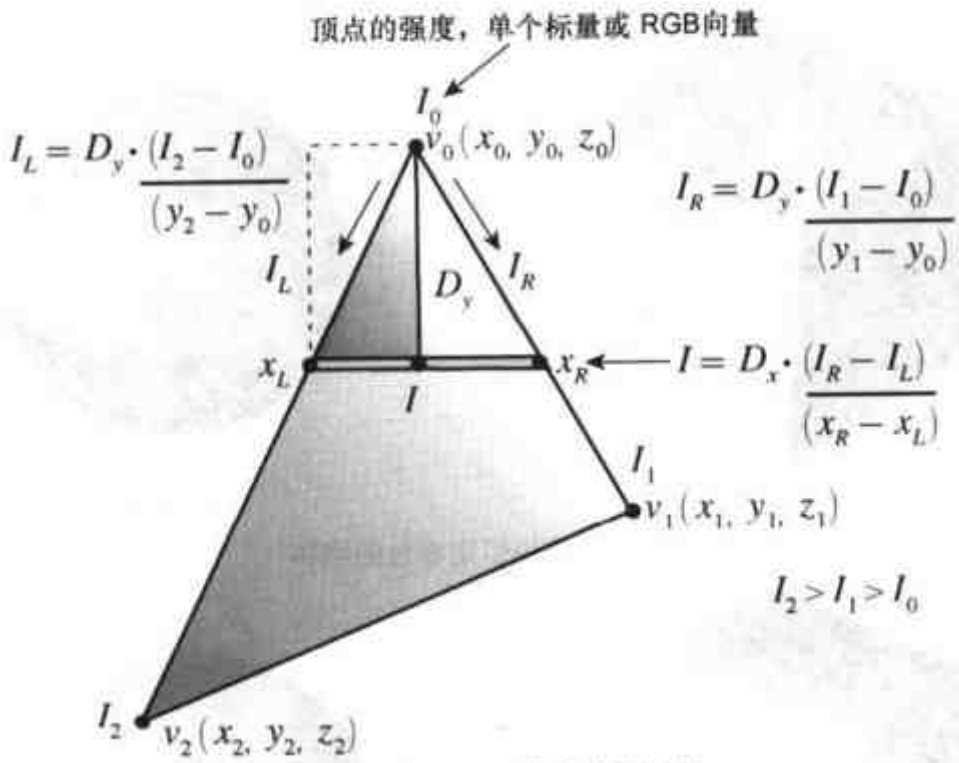


图 8.38　Gouraud 着色中的插值

读者对此应非常熟悉，它是一种标准插值算法，被用于从颜色到纹理图等各种东西的插值。本书将大量使用这种插值算法。下一章讨论仿射几何学（affine）纹理映射时，将深入探讨这种插值算法，如果读者现在还不太明白，到那时将会明白。

8.3.8　Phong 着色概述

Gouraud 着色是一种事实标准，市面上 99%的图形加速器都使用这种方法。它看起来很不错，但也存在没有考虑透视的缺点。Phong 着色解决了这种问题，它对每个像素执行着色计算。它不是在屏幕空间中通过插值来计算多边形中每个点的颜色，而是在屏幕空间中通过插值来计算多边形中每个点的法线，然后根据这些法线来执行光照计算，如图 8.39 所示。Phong 着色的真实感比 Gouraud 着色强些。它提高了镜面反射效果，使用 Gouraud 着色无法得到明显的镜面反射区域，因为它只是沿扫描线从一个值增加（降低）到另一个值，而 Phong 着色是针对像素的，每个像素都可能发生变化。

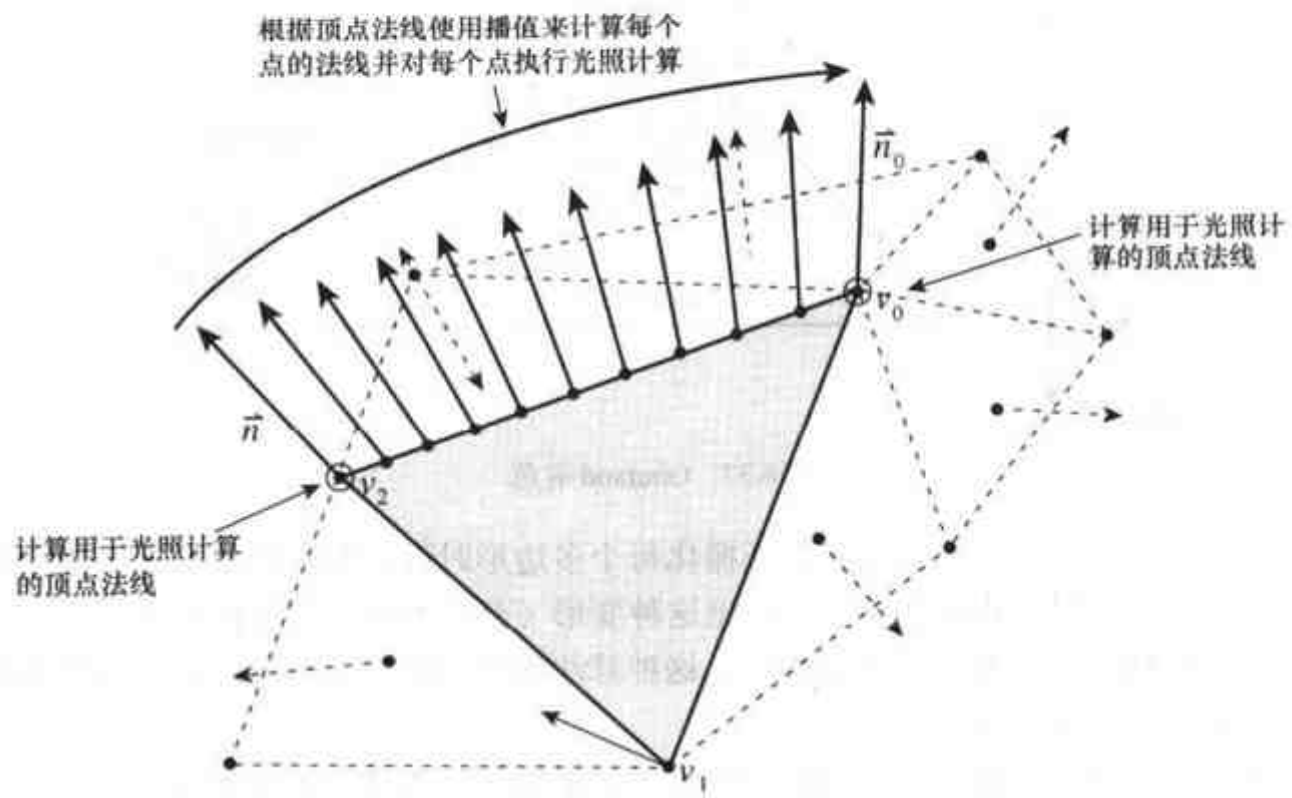


图 8.39　Phong 顶点法线插值算法

Phong 着色的缺点是，使用硬件都可能难以实现，更不用说使用软件了，因此本书不实现它。但在介绍高级光照处理时，将使用一些技巧来实现类似于 Phong 着色的镜面反射区域。

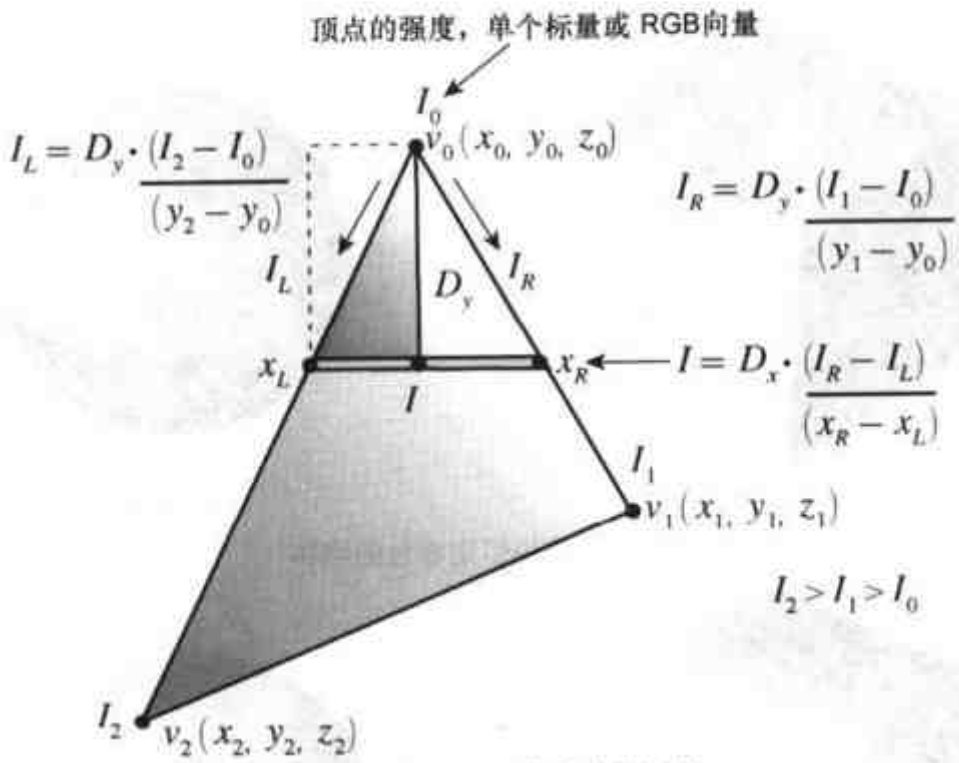


图 8.38　Gouraud 着色中的插值

读者对此应非常熟悉，它是一种标准插值算法，被用于从颜色到纹理图等各种东西的插值。本书将大量使用这种插值算法。下一章讨论仿射几何学（affine）纹理映射时，将深入探讨这种插值算法，如果读者现在还不太明白，到那时将会明白。

8.3.8　Phong 着色概述

Gouraud 着色是一种事实标准，市面上 99%的图形加速器都使用这种方法。它看起来很不错，但也存在没有考虑透视的缺点。Phong 着色解决了这种问题，它对每个像素执行着色计算。它不是在屏幕空间中通过插值来计算多边形中每个点的颜色，而是在屏幕空间中通过插值来计算多边形中每个点的法线，然后根据这些法线来执行光照计算，如图 8.39 所示。Phong 着色的真实感比 Gouraud 着色强些。它提高了镜面反射效果，使用 Gouraud 着色无法得到明显的镜面反射区域，因为它只是沿扫描线从一个值增加（降低）到另一个值，而 Phong 着色是针对像素的，每个像素都可能发生变化。

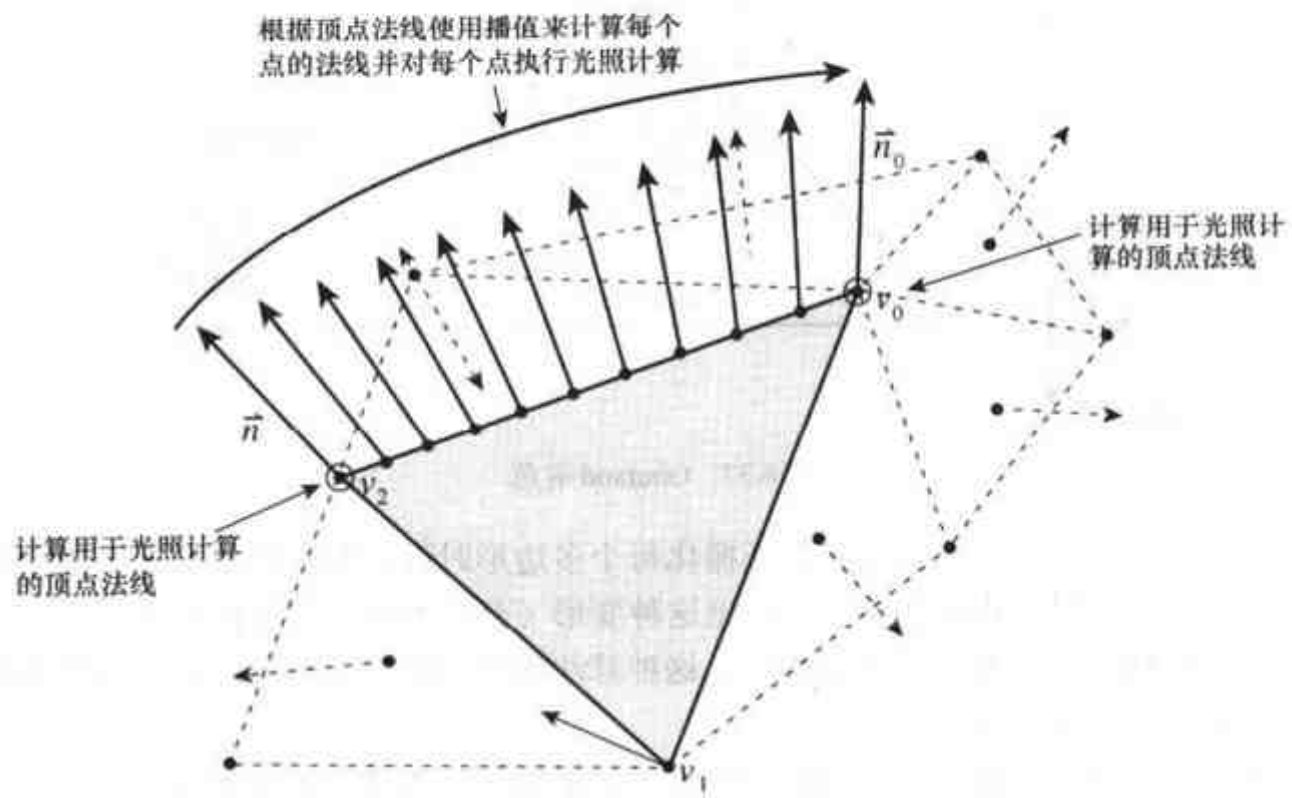


图 8.39　Phong 顶点法线插值算法

Phong 着色的缺点是，使用硬件都可能难以实现，更不用说使用软件了，因此本书不实现它。但在介绍高级光照处理时，将使用一些技巧来实现类似于 Phong 着色的镜面反射区域。

列表中对多边形进行排序，并相应地移动多边形在渲染列表中的位置。另一种方法是，新建一个列表，其中的索引/指针指向渲染列表中的多边形，然后对该列表进行排序。第一种方法看起来更容易，但移动实际数据的效率较低。

另外，渲染列表中可能包含被禁用的多边形，没有必要对其进行排序。因此，我们将创建一个指针数组，其中的每个指针都指向渲染列表中的一个多边形；然后对指针数组进行排序，并渲染指针数组而不是渲染列表本身。图 8.43 说明了这种方法。

从原来的渲染列表数据结构可知，它正是以这种方式实现的。

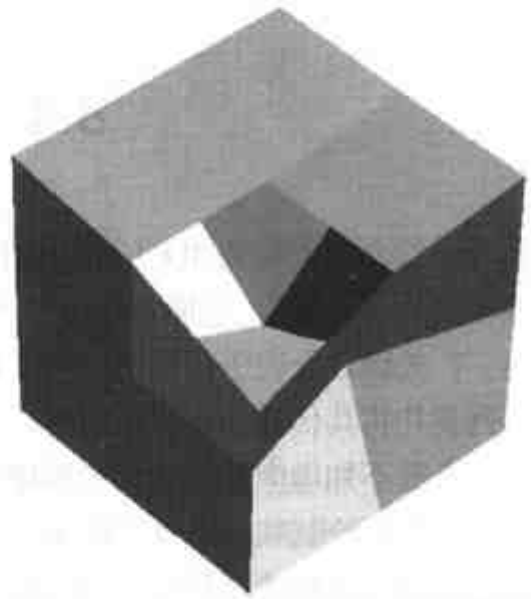


图 8.42　凹多面体导致画家算法遇到大量的异常情形

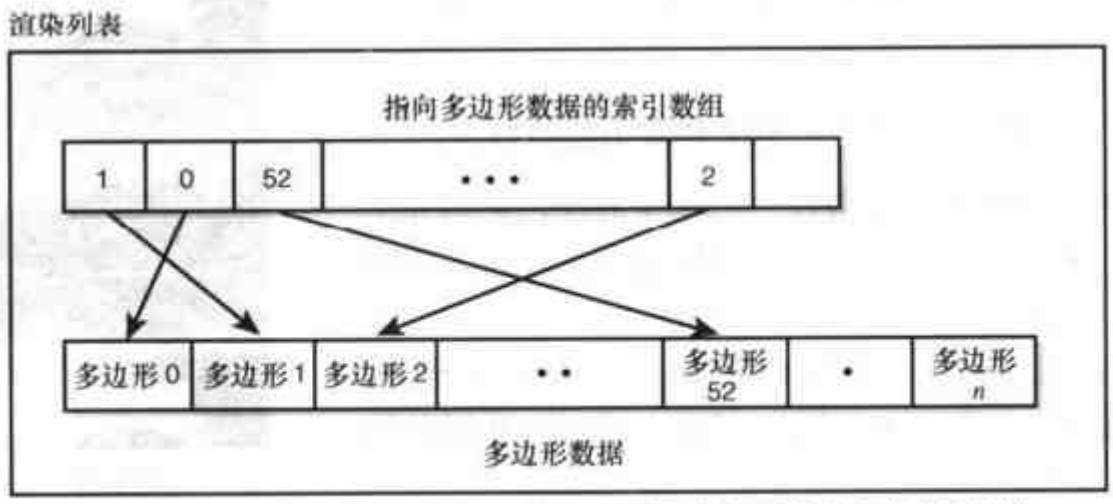


图 8.43　根据指向多边形的索引数组而不是多边形数据本身进行渲染

```
// 这是一个指针数组
typedef struct RENDERLIST4DV1_TYP
{
int state; // 渲染列表的状态
int attr;  // 渲染列表属性
// 渲染列表时一个指针数组
//其中每个指针都指向一个自包含的、可渲染的多边形面 POLYF4DV1
POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];

// 为避免每帧都为多边形分配和释放内存空间
// 多边形面被存储在 POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS]中
int num_polys; // 渲染列表中的多边形数

} RENDERLIST4DV1, *RENDERLIST4DV1_PTR;
```

从上述粗体代码可知，渲染列表实际上就是一个间接数组，可以使用它来排序。因此我们只需要排序算法，这里使用内置的快速排序算法，其函数原型如下：

```
void qsort(void *base,    // 目标数组的起始位置
     size_t num,        // 数组的长度（单位为元素）
     size_t width,      // 元素的长度，单位为字节
     int (cdecl *compare )(const void *elem1, // 比较函数
     const void *elem2 ) );  // 和要对其内容进行比较的指针
```

提示：快速排序的复杂度为 O(n*log n)，它是一种递归排序算法：将列表划分成几部分，对各个部分进行排序，然后将它们重新组装起来。

上述函数看似使用起来很简单，但涉及到指针时很麻烦。从比较函数的原型可知，它接受两个指针，它们分别指向要对其进行比较的内容。在这里，要比较的内容本身也是指针。这意味着在比较函数中，const void *是指向多边形面的指针的指针。这有些棘手，必须正确地进行处理。

另外，这里有必要讨论一些比较函数本身。我们要比较的是两个多边形的 z 值，但是哪种 z 值呢？

列表中对多边形进行排序，并相应地移动多边形在渲染列表中的位置。另一种方法是，新建一个列表，其中的索引/指针指向渲染列表中的多边形，然后对该列表进行排序。第一种方法看起来更容易，但移动实际数据的效率较低。

另外，渲染列表中可能包含被禁用的多边形，没有必要对其进行排序。因此，我们将创建一个指针数组，其中的每个指针都指向渲染列表中的一个多边形；然后对指针数组进行排序，并渲染指针数组而不是渲染列表本身。图 8.43 说明了这种方法。

从原来的渲染列表数据结构可知，它正是以这种方式实现的。

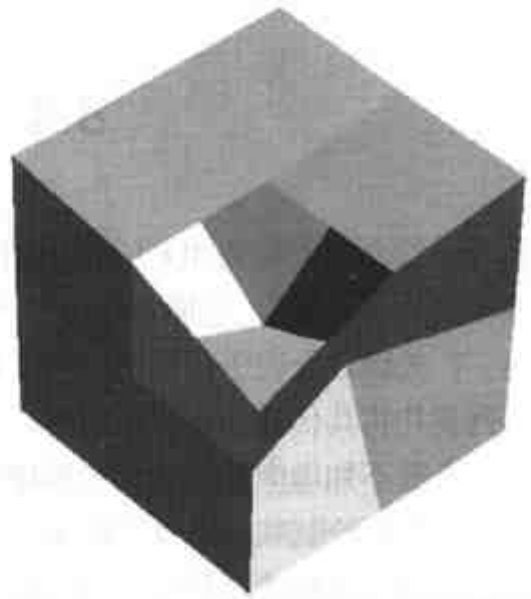


图 8.42　凹多面体导致画家算法遇到大量的异常情形

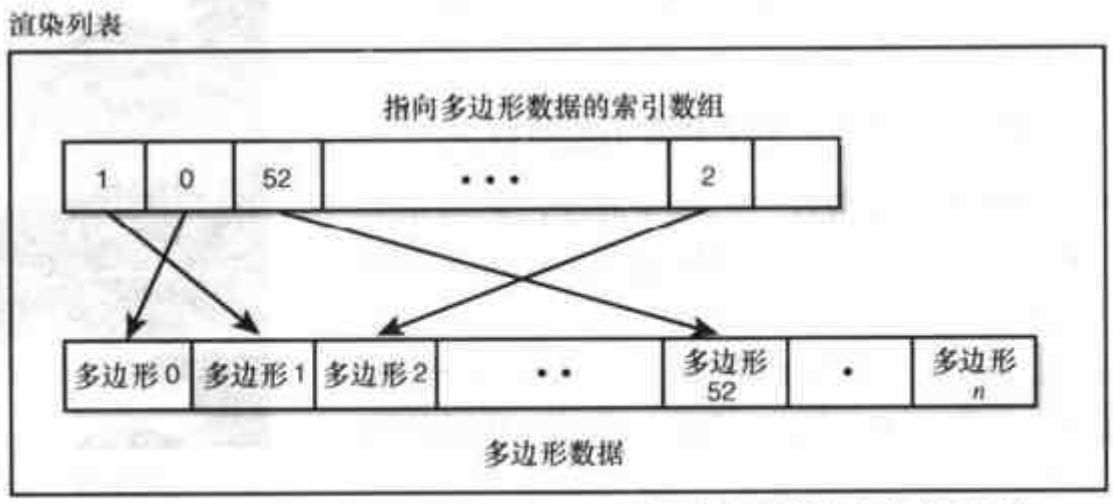


图 8.43　根据指向多边形的索引数组而不是多边形数据本身进行渲染

```
// 这是一个指针数组
typedef struct RENDERLIST4DV1_TYP
{
int state; // 渲染列表的状态
int attr;  // 渲染列表属性
// 渲染列表时一个指针数组
//其中每个指针都指向一个自包含的、可渲染的多边形面 POLYF4DV1
POLYF4DV1_PTR poly_ptrs[RENDERLIST4DV1_MAX_POLYS];

// 为避免每帧都为多边形分配和释放内存空间
// 多边形面被存储在 POLYF4DV1 poly_data[RENDERLIST4DV1_MAX_POLYS]中
int num_polys; // 渲染列表中的多边形数

} RENDERLIST4DV1, *RENDERLIST4DV1_PTR;
```

从上述粗体代码可知，渲染列表实际上就是一个间接数组，可以使用它来排序。因此我们只需要排序算法，这里使用内置的快速排序算法，其函数原型如下：

```
void qsort(void *base,    // 目标数组的起始位置
     size_t num,        // 数组的长度（单位为元素）
     size_t width,      // 元素的长度，单位为字节
     int (cdecl *compare )(const void *elem1, // 比较函数
     const void *elem2 ) );  // 和要对其内容进行比较的指针
```

提示：快速排序的复杂度为 O(n*log n)，它是一种递归排序算法：将列表划分成几部分，对各个部分进行排序，然后将它们重新组装起来。

上述函数看似使用起来很简单，但涉及到指针时很麻烦。从比较函数的原型可知，它接受两个指针，它们分别指向要对其进行比较的内容。在这里，要比较的内容本身也是指针。这意味着在比较函数中，const void *是指向多边形面的指针的指针。这有些棘手，必须正确地进行处理。

另外，这里有必要讨论一些比较函数本身。我们要比较的是两个多边形的 z 值，但是哪种 z 值呢？

这可以是三角形 3 个顶点的最大 z 值、最小 z 值或平均 z 值。由于对哪种 z 值进行比较更合适随具体情况而异，因此这里通过标记来指定使用哪种 z 值。排序算法根据标记来决定使用哪种 z 值进行比较，其代码如下：

```
void Sort_RENDERLIST4DV1(RENDERLIST4DV1_PTR rend_list, int sort_method)
{
// 这个函数根据 z 值对渲染列表中的多边形进行排序
// 具体的排序方式由参数 sort_method 指定
switch(sort_method)
{
case SORT_POLYLIST_AVGZ:  //  根据所有顶点的平均 z 值进行排序
     {
     qsort((void *)rend_list->poly_ptrs,
            rend_list->num_polys, sizeof(POLYF4DV1_PTR),
            Compare_AvgZ_POLYF4DV1);
     } break;
case SORT_POLYLIST_NEARZ: // 根据最小 z 值进行排序
     {
     qsort((void *)rend_list->poly_ptrs,
            rend_list->num_polys, sizeof(POLYF4DV1_PTR),
            Compare_NearZ_POLYF4DV1);
     } break;
case SORT_POLYLIST_FARZ:  //  根据最大 z 值进行排序
     {
     qsort((void *)rend_list->poly_ptrs,
            rend_list->num_polys, sizeof(POLYF4DV1_PTR),
            Compare_FarZ_POLYF4DV1);
     } break;

default: break;

} // end switch

} // end Sort_RENDERLIST4DV1
```

排序标记如下：

```
#define SORT_POLYLIST_AVGZ  0 //按平均 z 值排序
#define SORT_POLYLIST_NEARZ 1 //按最小 z 值排序
#define SORT_POLYLIST_FARZ  2 //按最大 z 值排序
```

最后，下面是一个范例比较函数（对平均 z 值进行比较）：

```
int Compare_AvgZ_POLYF4DV1(const void *arg1, const void *arg2)
{
// 这个函数对两个多边形的平均 z 值进行比较
// 深度排序算法将使用这个函数
float z1, z2;
POLYF4DV1_PTR poly_1, poly_2;
// 解除多边形指针引用
poly_1 = *((POLYF4DV1_PTR *)(arg1));
poly_2 = *((POLYF4DV1_PTR *)(arg2));

// 计算第一个多边形的平均 z 值
z1 = (float)0.33333*(poly_1->tvlist[0].z + poly_1->tvlist[1].z + poly_1-
>tvlist[2].z);

// 计算第二个多边形的平均 z 值
z2 = (float)0.33333*(poly_2->tvlist[0].z + poly_2->tvlist[1].z + poly_2-
>tvlist[2].z);
// 对 z1 和 z2 进行比较，以便将多边形按 z 值降序排列

if (z1 > z2)
   return(-1);
else
   if (z1 < z2)
return(1);
else
```

```
    return(0);
} // end Compare_AvgZ_POLYF4DV1
```

最后一个问题是，应该在什么地方调用排序算法？这很重要——必须根据相机坐标进行排序，因为此时所有多边形都处于投影前的最终位置。因此，需要在世界坐标到相机坐标变换之后和相机坐标到透视坐标变换之前进行排序，如下所示：

```
// 背面消除
Remove_Backfaces_RENDERLIST4DV1(&rend_list, &cam);

// 一次性对整个场景执行光照计算
Light_RENDERLIST4DV1_World16(&rend_list, &cam, lights, 3);

// 执行世界坐标到相机坐标变换
World_To_Camera_RENDERLIST4DV1(&rend_list, &cam);
// 对多边形列表进行排序
Sort_RENDERLIST4DV1(&rend_list, SORT_POLYLIST_AVGZ);

// 执行相机坐标到透视坐标变换
Camera_To_Perspective_RENDERLIST4DV1(&rend_list, &cam);

// 执行屏幕变换
Perspective_To_Screen_RENDERLIST4DV1(&rend_list, &cam);
```

有关这种算法的使用范例，请参阅 DEMOII8_6.CPP¦EXE（8 位版本为 DEMOII8_6_8b.CPP¦EXE），图 8.44 是这个演示程序运行时的屏幕截图。正如读者看到的，现在多边形被正确地排序，场景看起来是正确的（从很大程度上说是这样的）。请读者尝试在游戏世界中漫游（控制方法与前一个演示程序相同），看看是否会出现画家算法不灵的情形。另外，尝试修改排序算法，从根据平均 z 值排序改为根据最大或最小 z 值进行排序。

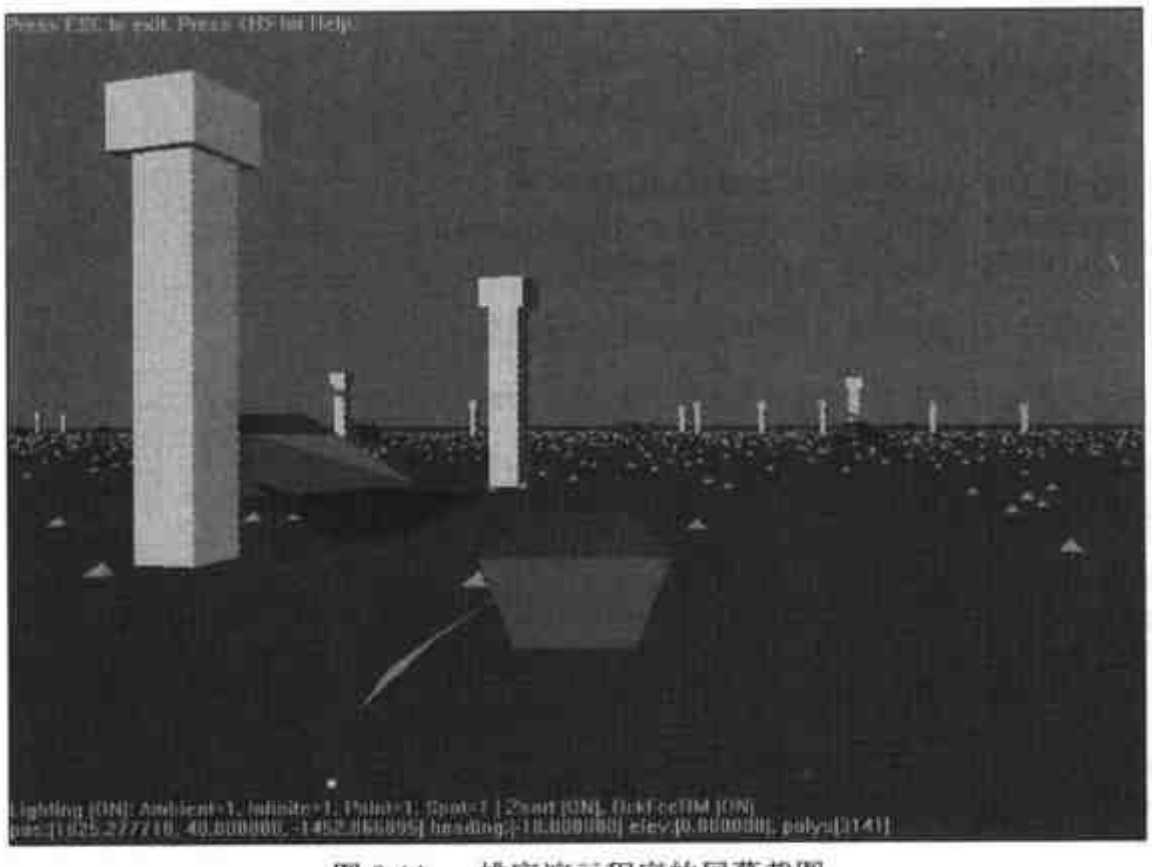


图 8.44　z 排序演示程序的屏幕截图

提示：处于简化的目的，这里对渲染列表中的多边形进行排序，但这并不一定是提供正确渲染顺序的最快方式，这取决于数据集。也可以分别对每个物体的多边形进行排序，然后根据包围球决定将物体插入到渲染列表的什么位置。换句话说，只要物体是凸形的且彼此不重叠，便可以在将物体插入渲染列表中时根据其中心位置确定其位置，来得到正确的渲染顺序，从而极大地减少排序时间。当然，仅对于太空游戏等室外游戏，这种方法才管用，在这种游戏中多边形位置不合适的情况较少，不会带来太大的麻烦。

8.5　使用新的模型格式

虽然到目前为止，模型格式一直不是什么大问题，但现在有必要支持更高级的模型格式。手工创建立方体和工兵（mech warrior）完全是两码事，后者通常无法使用手工方法创建。到目前为止我们一直使用的是.PLG/PLX 格式，现在需要支持其他一些文件格式。.PLG/PLX 格式很不错，它易于理解、操控和分析，但市面上没有任何建模程序支持这种格式，因此其用途有限。接下来编写另外两个分析器，用于分析一些最新的文件格式。

为此，需要在效率和教学效果（education）之间做出选择，当然是教学效果更重要。这里的要点在于，二进制分析器不那么直观，因此作者将使用 ASCII 文件格式，让读者能够查看和手工修改数据。这样，读者将能够自己编写二进制版本的加载函数。这里只编写两个新的加载函数：一个用于支持 3D Studio MAX ASCII 格式.ASC，另一个用于支持 Caligari trueSpace .COB（4.0 版以上）格式。附带光盘中提供了一些有关这两种格式的讨论文章。

这两种格式都有优缺点，但它们提供了这样的灵活性：使用工具创建 3D 模型（加载已有的模型），然后将它们以这些格式导出。当然，即使没有 Max 和 trueSpace，也可以使用转换程序来将工具保存的模型转换为其他格式。介绍文件加载函数之前，先来看一个用于帮助分析文本文件的 C++类（它更像一个 C 类）和辅助函数。

8.5.1　分析器类

无论是编写 ASCII 还是二进制格式的文件加载函数，在很大程度上说都是一个文件 I/O 练习。因此有必要编写一组让您能够加载文件、查找标记、剔除字符的函数。由于要处理的是 ASCII 文件，这里将编写一个具有如下功能的分析器：

- 打开/关闭文件；
- 读取一行文本；
- 从文本行中剔除字符；
- 获得由空格隔开的标记；
- 忽略注释字符后面的内容；
- 计算行数；
- 模式匹配。

图 8.45 是分析器系统的结构图。从图中可知，具备这些功能的分析器将对我们大有帮助。下面讨论模式匹配，这是分析器惟一有趣的方面。

需要一个模式匹配系统，以便能够在给定的文本行中搜索简单的正则表达式。例如，假设文本缓冲区中包含下列一行内容：

```
static char *buffer = "Polys: 10 Vertices: 20";
```

从中可以知道，“Ploys:”的后面是多边形数（一个整数），然后是“Vertices:”和顶点数（一个整数）。然而，由于其中的分号和其他不相关的字符，scanf()之类的函数无法对其进行分析，因此需要编写一个模式匹配函数来帮助分析。为此，可编写一个这样的模式匹配函数：

```
Pattern_Match(buffer, "['Polys: '] [i] ['Vertices: '] [i]"));
```

上述模式匹配函数依次查找并使用字符串“Polys:”、一个整数、字符串“Vertices:”和一个整数。介绍分析器类后，将讨论模式匹配语言。

还需要编写能够完成剔除字符、替换字符等任务的函数。基本上，我们需要一组比 C/C++提供的更健壮的字符串操作。这样的现成类有很多，但如果要使用引擎和类，可以自己编写 MOD——读者之所以阅读本书是想知道所以然。

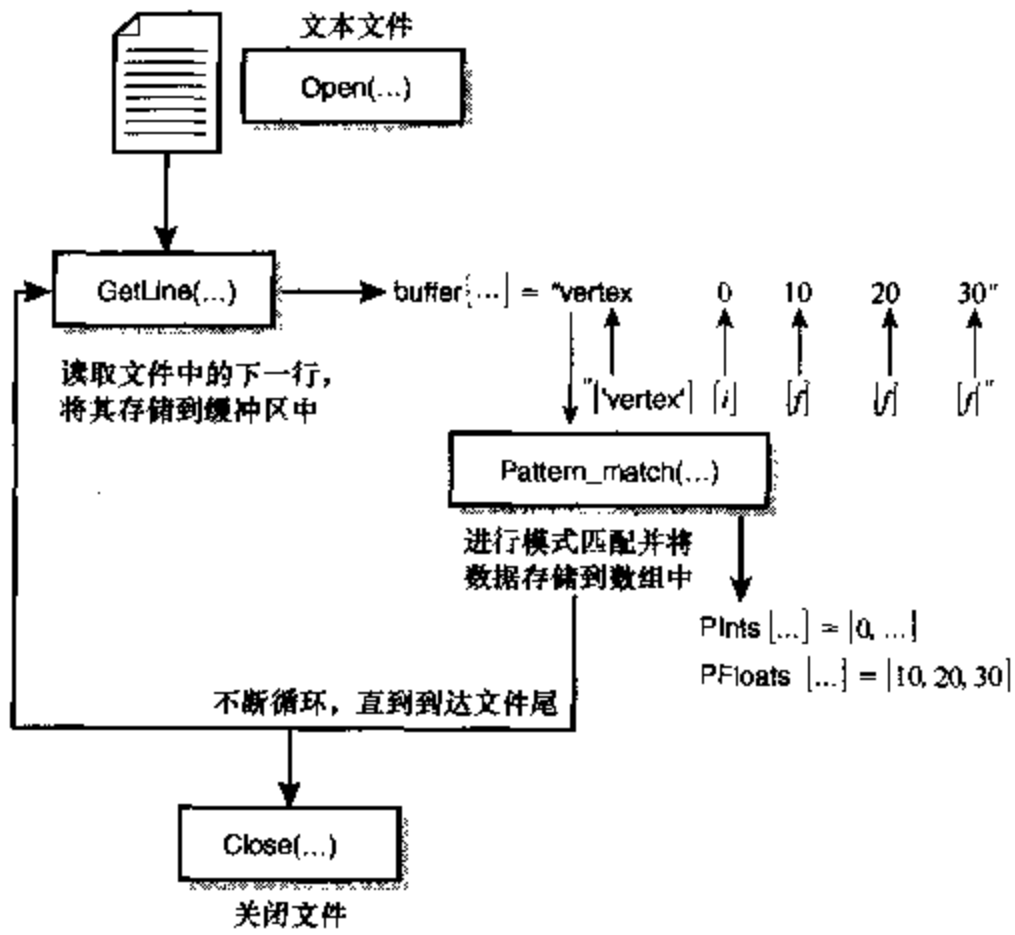


图 8.45　分析器系统的结构图

下面是分析器类的定义：

```
// parser class ////////////////////////////////////////////////
class CPARSERV1
{
public:
// 构造函数 ////////////////////////////////////////////////////
CPARSERV1();
// 析构函数 ////////////////////////////////////////////////////
~CPARSERV1() ;
// 重置文件系统 ////////////////////////////////////////////////
int Reset();
// 打开文件 ////////////////////////////////////////////////////
int Open(char *filename);
// 关闭文件 ////////////////////////////////////////////////////
int Close();
// 读取一行 ////////////////////////////////////////////////////
char *Getline(int mode);
// 设置注释字符串//////////////////////////////////////////////
int SetComment(char *string);
// 查找模式 ////////////////////////////////////////////////////
int Pattern_Match(char *string, char *pattern, ...);

// 变量声明 ////////////////////////////////////////////////////
public:
FILE *fstream;                          // 文件指针
char buffer[PARSER_BUFFER_SIZE];        // 缓存区
int  length;                            // 当前行长度
int  num_lines;                         // 处理了多少行
char comment[PARSER_MAX_COMMENT];       // 单行注释字符串

// 模式匹配参数存储空间
// 函数 pattern()返回时，匹配的内容将存储在这些数组中
char  pstrings[PATTERN_MAX_ARGS][PATTERN_BUFFER_SIZE]; //任何字符串
   int num_pstrings;
   float pfloats[PATTERN_MAX_ARGS];     // 任何浮点数
   int num_pfloats;

   int pints[PATTERN_MAX_ARGS];         //任何整数
```

```
    int num_pints;

}; // end CLASS CPARSERV1 ////////////////////////////////////////////////
typedef CPARSERV1 *CPARSERV1_PTR;
```

正如前面指出的，这是一个非常简单的 C++类。作者讨厌在教学中使用 C++，因为光看 C++代码可能根本不知道其含义，因此作者总是尽可能少用 C++。下面来看一下其中的方法。

正如读者看到的，这个分析器系统非常简单。类中包含以下函数：

- int Open（char *filename）——打开传入的文件路径指定的文件；如果成功则返回 1，否则返回 0；
- int Reset()——重置整个分析器，关闭所有的文件，并重置所有的变量；
- in Close()——调用函数 Reset()；
- char *Getline(int mode)——文件被打开后，该函数读取文件中的下一行；如果有下一行，则返回该行，否则返回 NULL。其中操作模式的取值如下：

```
#define PARSER_STRIP_EMPTY_LINES      1   // 删除空行
#define PARSER_LEAVE_EMPTY_LINES      2   // 保留空行
#define PARSER_STRIP_WS_ENDS          4   // 删除行尾的空格
#define PARSER_LEAVE_WS_ENDS          8   // 保留行尾的空格
#define PARSER_STRIP_COMMENTS         16  // 删除注释
#define PARSER_LEAVE_COMMENTS         32  // 保留注释
```

函数 Getline()返回指向当前行的指针，该行将被存储到类变量 buffer 中。

- int SetComment（char *string）——用于设置注释字符串，注释行以这样的字符串打头。可以包含除空格之外的任何字符，最多包含 16 个字符。在大多数情况下，字符串“#”、“;”和“//”是不错的注释标记，默认为“#”。
- int Pattern_Match（char *string, char *pattern, ...）——这个函数是所有函数中最复杂的，它在传入的字符串中查找与传入的模式匹配的子串。虽然 Pattern_Match()是分析器类的一个方法，但它允许您传入任何字符串，在其中查找模式匹配。表 8.3 解释了模式匹配语言。

表 8.3　　模式匹配语言

模式	含义	模式	含义
[i]	与整数匹配	[s<d]	与长度小于 d 个字符的任何字符串匹配
[f]	与浮点数匹配	[s>d]	与长度大于 d 个字符的任何字符串匹配
[s=d]	与包含 d 个字符的任何字符串匹配	['ssss...s']	与单引号中的字符串匹配

例如，要匹配下列内容：

```
"Vertex 3 34.56 23.67 10.90"
```

可使用模式"['Vertex'] [f] [f] [f]"。但是，匹配的字符串将被存储到哪里呢？在分析器类的末尾，有一些用粗体表示的变量和数组。例如，匹配浮点数时，浮点数本身和浮点数数目将被存储到下述变量中：

```
int pfloats[PATTERN_MAX_ARGS];
int num_pfloats;
```

对于前面的模式匹配范例，num_floats = 3，pfloats[] = {34.56, 23.67, 10.90}。

注意：读者可能注意到了函数 Pattern_Match()的参数列表。以后将通过该参数列表返回上述数组中的值，这里之所以没有这样做是因为现在使用这些数组是可行的，以后读者将发现这是一种不错的特性，进而实现它。

模式匹配函数存在两个缺点。它只能处理由空格分开的标记，对于下面这样的字符串：

```
"Vertex 3: 20.9, 90.8, 100.3"
```

这个模式匹配函数不管用。但辅助函数可帮助解决这种问题。下面介绍用于简化工作的辅助函数。

8.5.2 辅助函数

下面是一个小型字符串处理辅助函数库，它们位于 T3DLIB6.CPP|H 中。这些函数可单独使用，也可和分析器一起使用。

首先介绍字符剔除函数：

```
// 从字符串中删除字符
int StripChars(char *string_in, char *string_out, char *strip_chars,
               int case_on=1);
```

这个函数接受要对其进行字符剔除的字符串、用于存储处理结果的字符串、要剔除的字符以及大小写标记作为参数。下面的例子从字符串中剔除“:”和“,”：

```
StripChars("Vertex 3: 20.9, 90.8, 100.3", buffer,":,", 1);
```

结果为“Vertex 3 20.9 90.8 100.3”。

接下来看替换字符的函数：

```
// 替换字符串中的字符
int ReplaceChars(char *string_in, char *string_out,
                 char *replace_chars, char rep_char, int case_on=1);
```

上述函数用于替换字符，它接受的参数与函数 StripChars()类似，但多了一个 char 参数，该参数用于指定替换字符。对于下面的字符串：

```
"Vertex #3: 20.9,90.8,100.3"
```

如果使用函数 StripChars()剔除其中的“#”、“,”和“:”，结果将如下：

```
"Vertex 320.990.8100.3"
```

这里的问题是，剔除这些字符后，字符串粘在了一起。而我们想使用空格将它们分开，为此可使用函数 ReplaceChars()：

```
ReplaceChars("Vertex #3: 20.9,90.8,100.3",
buffer,"#:,", ' ', 1);
```

结果为：

```
"Vertex  3  20.9 90.8 100.3"
```

这正是我们希望的。这样，便可以使用 Pattern_Match（buffer,"[i] [f] [f] [f]"）来匹配第一个整数和接下来的 3 个浮点数。

下面是一些删除空格字符的函数：

```
// 删除字符串开头的所有空格
char *StringLtrim(char *string);

// 删除字符串末尾的所有空格
char *StringRtrim(char *string);
```

接下来的两个函数分别判断字符串是否可以转换为浮点数和整数，如果可以，则返回转换后的值。它们自己执行模式匹配，如果传入的字符串不是浮点数或整数，这两个函数将分别返回 FLT_MIN 和 INT_MIN，这是 limits.h 中定义的错误。

```
// 将字符串转换为浮点数
float IsFloat(char *fstring);

//将字符串转换为整数
int IsInt(char *istring);
```

下面是函数 IsFloat()的代码，其中包含很多分析技巧：

```
float IsFloat(char *fstring)
{
```

```
// 检查字符串是否是浮点数，如果是则对其进行转换
// 否则返回 FLT_MIN
// 字符串要能被转换为浮点数，必须是下面这样的格式：
//[whitespace] [sign] [digits] [.digits] [ {d | D | e | E }[sign]digits]
char *string = fstring;
// [whitespace]
while(isspace(*string)) string++;

// [sign]
if (*string=='+'||*string=='-')string++;

// [digits]
while(isdigit(*string)) string++;

// [.digits]
if (*string =='.')
   {
string++;
while(isdigit(*string)) string++;
   }

// [ {d | D | e | E }[sign]digits]
if (*string =='e'||*string=='E'||*string=='d'||*string=='D')
   {
string++;
// [sign]
if (*string=='+'||*string=='-')string++;
// [digits]
while(isdigit(*string)) string++;
   }

// 经过上述处理后，如果到达了字符串末尾，则可以转换为浮点数
if (strlen(fstring) == (int)(string - fstring))
return(atof(fstring));
else
return(FLT_MIN);
} // end IsFloat
```

作者并不想讨论分析问题。作者讨厌它，并想避开它。这里介绍它只是为了避免给读者提供一个黑盒式分析器，然后编写神奇的函数来加载新的模型格式。现在，读者应对这些函数的功能有了大概的了解。另外，辅助函数确实很有用！作为一个例子，演示程序 DEMOII8_7.CPP|EXE（图 8.46 是其屏幕截图）让您能够加载文本文件，对其进行分析，在其中查找模式。

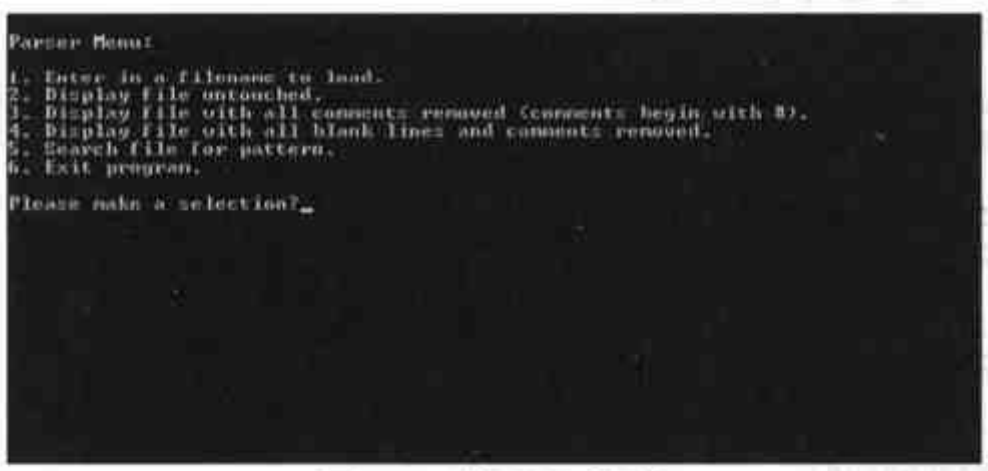


图 8.46　分析器测试程序

在附带光盘中，本章对应的目录中包含两个文本文件：TEST1.TXT 和 TEST2.TXT，其内容如下：

```
# test 1
object name: tank
num vertices: 4

vertex list:
0 10 20
```

```
5 90 10
3 4 3
1 2 3

end object # this is a comment!

# test 2
This is a line...
# I am commented out!
And this a 3rd...
```

请运行该程序，加载文件名，然后打印各种分析函数的处理结果。尝试在文件 TEST1.TXT 使用模式“[i] [i] [i]”。查看该程序的代码，它演示了如何使用分析器和辅助函数。

8.5.3 3D Studio MAX ASCII 格式.ASC

3D Studio MAX ASCII 格式.ASC 是一种非常简单的格式，它包含顶点和多边形信息以及每个多边形的材质。不幸的是，它不包含纹理信息和反射信息，因此对于纹理化物体，.ASC 格式不够强大。但这种格式比.PLG 好得多，实际上，99%的 3D 建模程序都支持以这种格式导出模型，因此使用这种格式可以简化工作。另一方面，在.ASC 格式中，无法选择多边形的着色模型，因为它对着色模型的支持很有限。有关文件格式的细节，请参阅第 6 章。下面是一个表示标准立方体的.ASC 文件：

```
Ambient light color: Red=0.3 Green=0.3 Blue=0.3
Named object: "Cube"
Tri-mesh, Vertices: 8     Faces: 12
Vertex list:
Vertex 0:  X:-1.000000     Y:-1.000000     Z:-1.000000
Vertex 1:  X:-1.000000     Y:-1.000000     Z:1.000000
Vertex 2:  X:1.000000     Y:-1.000000     Z:-1.000000
Vertex 3:  X:1.000000     Y:-1.000000     Z:1.000000
Vertex 4:  X:-1.000000     Y:1.000000     Z:-1.000000
Vertex 5:  X:1.000000     Y:1.000000     Z:-1.000000
Vertex 6:  X:1.000000     Y:1.000000     Z:1.000000
Vertex 7:  X:-1.000000     Y:1.000000     Z:1.000000
Face list:
Face 0:    A:2 B:3 C:1 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 1:    A:2 B:1 C:0 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 2:    A:4 B:5 C:2 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 3:    A:4 B:2 C:0 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 4:    A:6 B:3 C:2 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 5:    A:6 B:2 C:5 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 6:    A:6 B:7 C:1 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 7:    A:6 B:1 C:3 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 8:    A:6 B:5 C:4 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 9:    A:6 B:4 C:7 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
```

```
Face 10:    A:1 B:7 C:4 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
Face 11:    A:1 B:4 C:0 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
```

要读取这种文件格式，需要编写一个执行下述步骤的分析器：

第 1 步　读取物体名、顶点数和多边形数；

第 2 步　读取顶点列表；

第 3 步　读取每个多边形的定义及其 RGB 材质颜色信息。

这种文件格式并不复杂，分析起来很容易。其缺点是，不能指定多边形是单面还是双面的、不支持纹理映射信息和光照模型信息等；实际上这种格式只提供了每个多边形的颜色。因此我们的阅读器必须对诸如光照模型等方面进行假设。就现在而言，我只是想编写一个这样的阅读器：在 8 位或 16 位图形模式下加载.ASC 文件，然后相应地设置数据结构的内容，并将光照等方面的信息设置为默认值。在此之前，先来看一下多边形的定义，确保读者掌握了这方面的内容：

```
Face 10:    A:1 B:7 C:4 AB:1 BC:1 CA:1
Material: "r255g255b255a0"
Smoothing:  1
```

这是一个面（10）。它由顶点 A、B 和 C 构成，这些顶点的实际编号分别是 1、7 和 4。文本 AB、BC 和 CA 描述了多边形顶点的环绕方向（我们没有使用），“smoothing”指出是否对多边形法线进行平均（也没有使用）。有关这种格式的完整描述，请参阅附带光盘中的目录 3D_DOSC\File Formats\。

下面是加载函数的原型：

```
int Load_OBJECT4DV1_3DSASC(OBJECT4DV1_PTR obj,   // 执行物体的指针
                           char *filename,       // ASC 文件的名称
                           VECTOR4D_PTR scale, // 初始缩放因子
                           VECTOR4D_PTR pos,     // 初始位置
                           VECTOR4D_PTR rot,     // 初始旋转角度
                           int vertex_flags)     // 顶点重新排序标记
```

这个函数的原型与.PLG 加载函数几乎相同，只是多了参数 vertex_flags。这个参数让您在加载时对物体进行控制。使用建模程序创建物体时，坐标常常会被反转（invert）或交换，环绕顺序导致多边形方向朝后，等等。这个参数让您能够在加载物体是对这些方面进行修改。例如，我们的引擎使用左手坐标系，而建模程序使用的可能是右手坐标系，在这种情况下必须反转 z 轴。参数 vertex_flags 可以有下列值通过逻辑 OR 运算组合而成：

```
#define VERTEX_FLAGS_INVERT_X          1 // 反转 x 轴
#define VERTEX_FLAGS_INVERT_Y          2 // 反转 y 轴
#define VERTEX_FLAGS_INVERT_Z          4 // 反转 z 轴
#define VERTEX_FLAGS_SWAP_YZ           8 // 从 RHS 变换为 LHS
#define VERTEX_FLAGS_SWAP_XZ           16
#define VERTEX_FLAGS_SWAP_XY           32
#define VERTEX_FLAGS_INVERT_WINDING_ORDER 64 // 反转环绕顺序
在环绕顺序被颠倒时，这样加载物体 CUBE.ASC：
int Load_OBJECT4DV1_3DSASC(&obj, "CUBE.ASC",
                           NULL, NULL, NULL,
                           VERTEX_FLAGS_INVERT_WINDING_ORDER);
```

警告：这个阅读器只能读取 ASCII .ASC 文件。不要使用它来读取二进制文件，否则将崩溃。

该函数的代码太长，这里无法列出它们，请参阅附带光盘中的 T3DLIB.CPP。这个函数的代码之所以很长，是由于包含大量的错误处理代码，虽然这些错误在文件被正确加载时根本不会发生。

有关使用这个函数的例子，请参阅 DEMOII8_8.CPP|EXE，图 8.47 是这个演示程序的屏幕截图。它让您能够通过菜单从磁盘加载任何.ASC 物体，您必须指定文件的完整路径。下面是作者创建的一些物体，它们存储在目录 T3DIICHAP08\中：



图 8.47　使用 3D Studio .ASC 加载函数的演示程序

- SHPERE01.ASC（坐标 YZ 互换，环绕顺序反转）；
- CAR01.ASC（坐标 YZ 互换，环绕顺序反转）；
- HAMMER02.ASC。

警告：不能加载这样的物体：其顶点或多边形数目超过了引擎的处理能力。当前最大顶点数和多边形数都为 1024：

```
#define OBJECT4DV1_MAX_VERTICES   1024
#define OBJECT4DV1_MAX_POLYS      1024
```

这些#define 位于库模块 T3DLIB5.H 中。

另外，文件加载对话框中包含让您能够对 YZ 坐标进行互换以及反转环绕顺序的复选框。要正确地加载物体，可能需要设置这些复选框。

8.5.4　TrueSpace ASCII.COB 格式

“决不要低估魔戒（magic ring）的威力”，这句话也适用于 trueSpace 5+。如果不是这样，根本不要考虑它。经过长久的思索和权衡后，作者发现 Caligari .COB 格式是除 Microsoft .X 文件外，市面上最好的静态文件格式。它功能强大，支持我们需要的所有信息，其中包括材质的概念。使用这种文件格式，只需要将合适的材质应用于物体，便可以选择应用于多边形的着色模型。另外，COB 格式还支持纹理等。当然，如果读者没有 trueSpace，要以.COB 格式导出模型，需要使用阅读器或转换程序（如 Polytrans）。作者不打算编写.3DS 阅读器。使用这种格式简直是一场灾难，我讨厌晦涩而过时的格式。如果读者要使用.3DS 格式，必须自己编写阅读器。事实上，这种格式也没有那么糟糕，只是它是二进制的，难以查看其内容。

虽然第 6 章讨论过.COB 格式（有关这种格式的详细分析，请参阅附带光盘的 3D_DOCS\File Formats\），这里还是来看一下一个简单物体的.COB 文件的内容。该文件名为 CLIGARI_COB_FORMAT_EXAMPLE01.COB，其描述的物体如图 8.49 所示。

这是在 trueSpace 中创建的一个立方体，其中一个面被贴上了纹理。下面是这个 ASCII 文件的内容，作者加上了一些空行。另外，重要的内容使用粗体，并删除了一些高级区段，如简要描述（thumbnail）和辐射度，因此详细的内容请参阅原文件。由于原文件过长，这里无法列出。下面来看看这种格式：

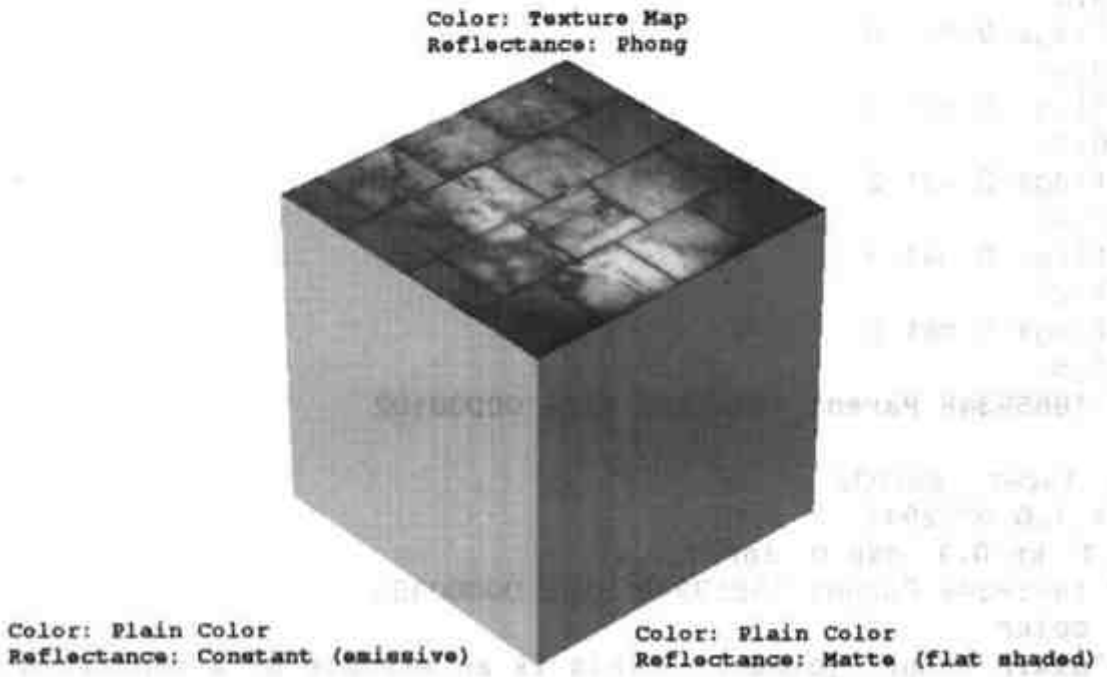


图 8.48　包含多种表面类型的立方体模型

```
Caligari V00.01ALH
PolH V0.08 Id 18661892 Parent 0 Size 00001060
Name Cube,1
center 0 0 0
x axis 1 0 0
y axis 0 1 0
z axis 0 0 1

Transform
1 0 0 1.19209e-007
0 1 0 0
0 0 1 -1
0 0 0 1

World Vertices 8
1.000000 -1.000000 0.000000
1.000000 -1.000000 2.000000
1.000000 -1.000000 0.000000
1.000000 -1.000000 2.000000
1.000000 1.000000 0.000000
1.000000 1.000000 0.000000
1.000000 1.000000 2.000000
1.000000 1.000000 2.000000

Texture Vertices 6
0.000000 0.000000
0.000000 1.000000
0.000000 0.000000
0.000000 1.000000
1.000000 0.000000
1.000000 1.000000

Faces 12
Face verts 3 flags 0 mat 4
<0,0> <1,1> <3,5>
Face verts 3 flags 0 mat 4
<0,0> <3,5> <2,4>
Face verts 3 flags 0 mat 0
<0,1> <2,5> <5,4>
Face verts 3 flags 0 mat 0
<0,1> <5,4> <4,0>
Face verts 3 flags 0 mat 1
<2,2> <3,3> <6,5>
Face verts 3 flags 0 mat 1
<2,2> <6,5> <5,4>
Face verts 3 flags 0 mat 0
```

```
<1,0> <7,1> <6,5>
Face verts 3 flags 0 mat 0
<1,0> <6,5> <3,4>
Face verts 3 flags 0 mat 2
<4,4> <5,0> <6,1>
Face verts 3 flags 0 mat 2
<4,4> <6,1> <7,5>
Face verts 3 flags 0 mat 3
<0,4> <4,2> <7,3>
Face verts 3 flags 0 mat 3
<0,4> <7,3> <1,5>
Mat1 V0.06 Id 18659348 Parent 18661892 Size 00000102
mat# 1
shader: phong  facet: auto32
rgb 0.00784314,1,0.0352941
alpha 1  ka 0.1  ks 0.1  exp 0  ior 1
ShBx V0.03 Id 18659349 Parent 18659348 Size 00000383
Shader class: color
Shader name: "plain color" (plain)  - this is an example of a non-textured material
Number of parameters: 1
colour: color (2, 255, 9)
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shader name: "constant" (constant)  this material is constant shaded, or emmisive
to us
Number of parameters: 0
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3

Mat1 V0.06 Id 18658628 Parent 18661892 Size 00000085
mat# 2
shader: phong  facet: auto32
rgb 1,0,0
alpha 1  ka 0.1  ks 0.1  exp 0  ior 1
ShBx V0.03 Id 18658629 Parent 18658628 Size 00000427
Shader class: color
Shader name: "plain color" (plain)  again a plain colored material
Number of parameters: 1
colour: color (255, 0, 0)
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shader name: "matte" (matte)  this material is flat shaded in our engine
Number of parameters: 2
ambient factor: float 0.1
diffuse factor: float 1
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3

Mat1 V0.06 Id 18657844 Parent 18661892 Size 00000101
mat# 3
shader: phong  facet: auto32
rgb 0.0392157,0.0117647,1
alpha 1  ka 0.1  ks 0.5  exp 0  ior 1
```

```
ShBx V0.03 Id 18657845 Parent 18657844 Size 00000522
Shader class: color
Shader name: "plain color" (plain)  again a colored material
Number of parameters: 1
colour: color (10, 3, 255)
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shader name: "plastic" (plastic)  this means use gouraud shading
Number of parameters: 5
ambient factor: float 0.1
diffuse factor: float 0.75
specular factor: float 0.5
roughness: float 0.1
specular colour: color (255, 255, 255)
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3

Mat1 V0.06 Id 18614788 Parent 18661892 Size 00000100
mat# 4
shader: phong  facet: auto32
rgb 1,0.952941,0.0235294
alpha 1  ka 0.1  ks 0.1  exp 0  ior 1
ShBx V0.03 Id 18614789 Parent 18614788 Size 00000515
Shader class: color
Shader name: "plain color" (plain)  a plain colored material
Number of parameters: 1
colour: color (255, 243, 6)
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shader name: "phong" (phong)  actually use a phong shader in our engine
Number of parameters: 5
ambient factor: float 0.1
diffuse factor: float 0.9
specular factor: float 0.1
exponent: float 3
specular colour: color (255, 255, 255)
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3

Mat1 V0.06 Id 18613860 Parent 18661892 Size 00000182
mat# 0
shader: phong  facet: auto32
rgb 1,0.952941,0.0235294
alpha 1  ka 0.1  ks 0.1  exp 0  ior 1
texture: 36D:\Source\models\textures\wall01.bmp
offset 0,0  repeats 1,1  flags 2
ShBx V0.03 Id 18613861 Parent 18613860 Size 00000658
Shader class: color
Shader name: "texture map" (caligari texture)  this means this material is a tex-
ture map
Number of parameters: 7
file name: string "D:\Source\models\textures\wall01.bmp"  here the texture
S repeat: float 1
```

```
T repeat: float 1
S offset: float 0
T offset: float 0
animate: bool 0
filter: bool 0
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shader name: "phong" (phong)  the shader for this material should be phong
Number of parameters: 5
ambient factor: float 0.1
diffuse factor: float 0.9
specular factor: float 0.1
exponent: float 3
specular colour: color (255, 255, 255)
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
END  V1.00 Id 0 Parent 0 Size       0
```

这种格式很容易理解。下面解释其中的一些细节，以便读者知道如何编写阅读器。读取.COB 文件的基本步骤如下：

第 1 步 读取文件开头的第一行。

第 2 步 读取名称、中心位置、x 轴、y 轴和 z 轴。这些内容定义了对物体的局部变换。这种变换必须应用于物体的顶点，因为出于精确度的考虑，导出 trueSpace 物体时，其顶点坐标是未经变换的。换句话说，当您在 trueSapce 中旋转或移动立方体时，建模程序记录变换并渲染物体，但保留原来的立方体——以免降低精确度。

第 3 步 读取变换行和随后的矩阵。这是局部坐标到世界坐标的变换矩阵，在大多数情况下，这是一个单位矩阵，但我们将支持应用这个矩阵以及前一个矩阵，以确保模型位于建模程序要将其放置到的位置，且处于正确的朝向。

第 4 步 读取 World Vertices 行，然后按 x、y、z 顺序读取顶点列表。

第 5 步 读取 Texture Vertices 行，然后读取每个顶点的<u，v>纹理坐标。在.COB 格式中，对纹理坐标进行了压缩。也就是说，一般而言每个三角形应该有 3 对纹理坐标，但.COB 格式好像重用了重复的（u，v）值，因此对于包含 100 个三角形的网格，大多数情况下并没有 300 对纹理坐标。采用的是什么样的压缩逻辑作者并不知道。但这无关紧要，只需将纹理坐标读入一个数组，然后将这些值存储到最终的纹理坐标数组中即可（我们的引擎中还没有纹理坐标数组，因此暂时不管它）。

注意：纹理坐标是要粘贴到多边形表面上的纹理中的样本点。在大多情况下，纹理的坐标系如图 8.49 所示。其中 u 表示 x 轴，v 表示 y 轴，同时 99% 的纹理都是方形的。因此，要将图中的纹理映射到一对三角形上，可以使用图中所示的纹理坐标。这将在本书后面详细介绍。

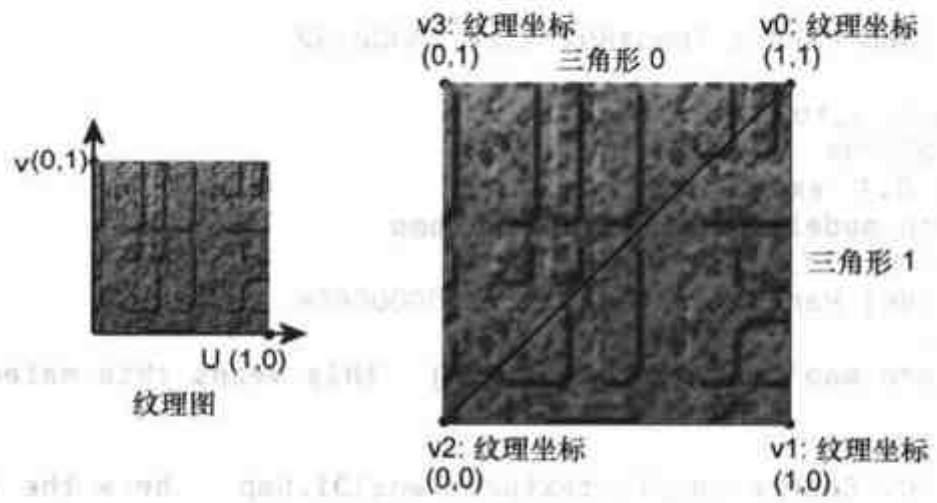


图 8.49 标准纹理坐标

第 6 步　读取 Face 行，然后读取每个多边形面。面可以有很多顶点，但我们将确保所有导出的面都只有 3 个顶点，因此 Face 行如下：

```
Face verts 3 flags ff mat mm
```

其中 *ff* 和 *mm* 分别是标记和材质号。标记无关紧要，不用管它；但材质号非常重要。材质号最小为 0，它指出了应将哪种材质应用于多边形面。分析文件时，必须记录材质号，以便后面读取材质时可以进行合理的安排。

接下来是多边形面本身，它是一组顶点索引，格式为<*v*1，*t*1><*v*2，*t*2>……第一个数字是三角形各个顶点的索引，第二个数字是纹理坐标。因此<1，2><9，5><3，8>的含义是，这个三角形是由顶点 1、9 和 3 组成，这些顶点的纹理坐标分别是 2、5 和 8，呈一一对应的关系。

第 7 步　现在问题变得有些棘手。我们必须读取材质（它们可能并不是按顺序排列的）。由于我们的引擎不是硬件引擎，因此不打算支持每种材质选项。需要以某种方式支持下列属性：

```
#define MATV1_ATTR_2SIDED                   0x0001
#define MATV1_ATTR_TRANSPARENT              0x0002
#define MATV1_ATTR_8BITCOLOR                0x0004
#define MATV1_ATTR_RGB16                    0x0008
#define MATV1_ATTR_RGB24                    0x0010

#define MATV1_ATTR_SHADE_MODE_CONSTANT      0x0020
#define MATV1_ATTR_SHADE_MODE_EMMISIVE      0x0020 // 别名
#define MATV1_ATTR_SHADE_MODE_FLAT          0x0040
#define MATV1_ATTR_SHADE_MODE_GOURAUD       0x0080
#define MATV1_ATTR_SHADE_MODE_FASTPHONG     0x0100
#define MATV1_ATTR_SHADE_MODE_TEXTURE       0x0200
```

很少建模程序在多边形列表或材质中提供了双面信息，因此通过读取模型文件无法知道多边形的双面属性。另外，所有的建模型程序都使用 RGB 值来表示材质属性，因此需要在加载模型时根据模式计算色深。在.COB 文件中只能指定上述以粗体显示的属性。幸运的是，.COB 格式的材质支持允许导出很多这样的内容，即可以使用它们来指定要将哪种光照模型用于多边形。

例如，假设我们创建了一个船舶模型，并希望引擎对其使用固定着色，而对其他物体使用 Gouraud 着色。为此，需要应用合适的材质，以便能够提取这样的信息。如果建模程序不支持某种导出这种信息的方式，必须采取更巧妙的方式。例如，可以这样规定，任何灰色多边形都是发射多边形，同时根据灰度，将颜色值设置为 0～15。

我们无需这样做，因为可以使用建模程序指定材质属性。然而，建模程序中的材质着色模式和我们所需的着色模式之间并非是 1:1 对应的，稍后将讨论这种问题。现在讨论实际编码。使用哪种建模程序无关紧要，只要将物体作为 trueSpace 4.0+ .COB 物体导出即可。编码方式如下。

首先，找到材质。材质都以类似于下面的内容打头（第二行是材质号）：

```
Mat1 V0.06 Id 142522724 Parent 18646020 Size 00000182
mat# 0
```

这是材质 0 的定义，使用材质 0 的多边形都将使用这种材质。接下来我们感兴趣的几行如下：

```
rgb 0.0392157,0.0117647,1
alpha 1  ka 0.1  ks 0.5  exp 0  ior 1
```

其中第一行是材质的颜色；第二行是 alpha 值、反射系数和其他几种信息，如表 8.4 所示。

表 8.4　　**.COB 材质中 Alpha 行的变量**

变量	含义	取值范围	变量	含义	取值范围
alpha	透明度	0～1	exp	镜面反射指数	0～1
ka	环境光反射系数	0～1	ior	折射率	（我们没有使用）
ks	镜面光反射系数	0～1			

下面介绍如何提取着色处理模型信息。在前面列出的.COB 文件中，每种材质定义中都有 4 行为粗体，如下所示：

```
Shader class: color
Shader name: 'plain color' (plain)
.
.
.
.
Shader class: reflectance
Shader name: "phong" (phong)
.
.
```

需要特别注意这些文本行。Shader class: color 和 Shader Class: reflectance 是进行编码时需要考虑的关键内容。确定多边形是普通彩色（plain color）多边形（即使用固定、恒定、Gouraud 或 Phong 着色）还是被贴上了纹理的多边形。

首先，需要知道着色模型。为此，只需找到这样的文本行：

```
Shader class: color
```

然后找到这样的文本行：

```
Shader name: "xxxxxx" (xxxxxx)
```

对于普通彩色多边形，上述两个 xxxxx 分别是“Plain color”和“plain”；而对于纹理，两个 xxxxx 分别是“texture map”和“caligari”。然后查找：

```
Shader class: reflectance
```

为确定着色类型，我们查找这样的文本行：

```
Shader name: "xxxxxx" (xxxxxx)
```

然后根据上述文本行中的值相应地设置着色系统，如表 8.5 所示。

表 8.5　　.COB shader 模型和引擎使用的 shader 模型之间的对应关系

.COB shader 模型	引擎使用的 shader 模型
“constant”	MATV1_ATTR_SHADE_MODE_CONSTANT
“matte”	MATV1_ATTR_SHADE_MODE_FLAT
“plastic”	MATV1_ATTR_SHADE_MODE_GOURAUD
“phong”	MATV1_ATTR_SHADE_MODE_FASTPHONG
“texture map”	MATV1_ATTR_SHADE_MODE_TEXTURE

注意：纹理映射必须同其他模式结合使用，以便对纹理进行着色。

在建模程序中，无法导出有关模型的信息。例如，如果使用 MAX 来创建游戏世界，如何指出门是门呢？这与前面的情形类似，虽然没那么糟糕，但不得不承认这确实是个问题。我们必须找到在这种文件格式中对有关模型的信息进行编码的方法。

鉴于读者已经茫然失措，下面简要地重申一下在 trueSpace 建立我们的.COB 阅读器能够读取的模型。首先，按通常那样建立物体模型，然后将材质应用于物体。如果希望多边形是普通彩色的，保留材质颜色“plain color”，而不要将其修改为“texture map”。然后在模型的反射部分，根据要采用何种着色方法，选择反射模型 constant、matte、plastic 或 Phong。当前，我们的引擎只支持固定着色和恒定着色。如果要给表面贴上纹理，只需像通常那样做即可，建模程序将把颜色设置为“texture”，并应用指定的纹理。然后，需要设置反射模型，指定如何对纹理执行着色。当然，我们的引擎现在还不支持纹理，因此不要尝试这样做。

至此，对.COB 文件格式的讨论就结束了，但如何处理材质呢？事实上，加载.COB 物体时，材质被加

载到材质库中，如图 8.50 所示。然而，这存在两个问题。首先，我们的引擎还没有使用材质，而只使用了光源。其次，两个物体使用的材质可能有相同的，因此需要考虑材质重用的问题。当然，由于之前使用的 3D 文件格式不支持材质，因此我们的引擎没有使用材质。

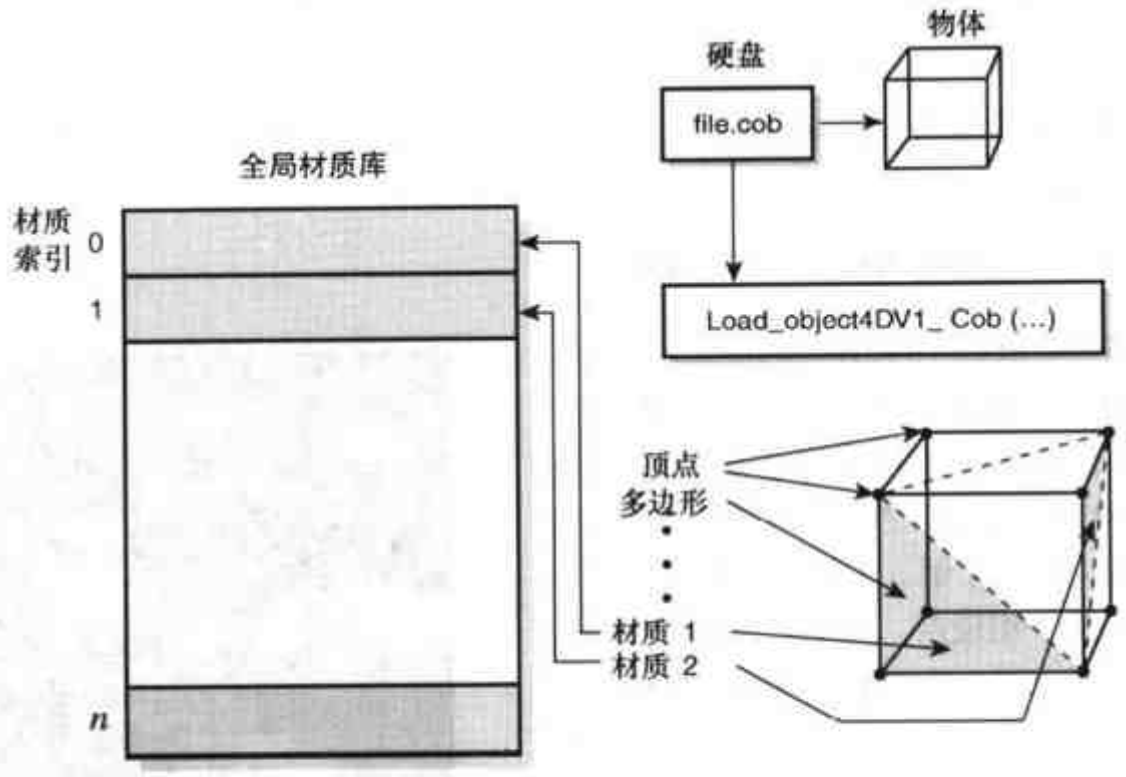


图 8.50　.COB 阅读器将材质加载到全局材质库中

警告：由于我们的引擎只能处理三角形，因此必须将所有.COB 物体导出为三角形，所以导出之前，必须将所有物体划分成三角形。

读者对.COB 格式有一定的了解后，下面来看一下文件加载函数的原型。除了新增了一些功能外，它与.ASC 加载函数几乎相同：

```
int Load_OBJECT4DV1_COB(OBJECT4DV1_PTR obj,   // 指向物体的指针
                char *filename,        // Caligari COB 文件名称
                VECTOR4D_PTR scale,    // 初始缩放因子
                VECTOR4D_PTR pos,      // 初始位置
                VECTOR4D_PTR rot,      // 初始旋转角度
                int vertex_flags)      // 环绕顺序反转和变换标记
```

惟一的不同之处在于，新增了参数 vertex_flags，可以将这个参数设置为下述值的逻辑 OR 组合，以启用局部变换或世界变换：

```
#define VERTEX_FLAGS_TRANSFORM_LOCAL        512   // 如果文件格式中包含局部变
                                                  //换，则执行这种变换
#define VERTEX_FLAGS_TRANSFORM_LOCAL_WORLD  1024  // 如果文件格式中包含世界变
                                                  //换，则执行这种变换
```

下面是一个加载.COB 物体的例子，其中参数 vertex_flags 的设置适用于大多数建模程序导出的.COB 物体：

```
Load_OBJECT4DV1_COB(&obj_tank, "D:/Source/models/cobs/hammer03.cob",
                    &vscale, &vpos, &vrot,
                    VERTEX_FLAGS_SWAP_YZ | VERTEX_FLAGS_TRANSFORM_LOCAL );
```

这里指定了缩放向量、位置向量和旋转向量，这些参数是可选的。

至此，对.COB 文件格式的讨论就结束了。有关这种文件格式的规范，请参阅附带光盘中的目录 3D_DOCS\File Formats\，各个函数的代码请参阅文件 T3DLIB6.CPP|EXE。有关使用该加载函数的例子，请参阅程序 DEMOII8_9.CPP|EXE，它与加载.ASC 文件的演示程序完全相同，只是加载的是.COB 文件。

另外，文件加载对话框中包含这样的复选框：让您能够将 YZ 坐标互换，反转环绕顺序，启用局部变换和世界变换。为正确地加载物体，可能需要设置这些复选框。最后，我们只为每个物体提供 1024 个顶点

和 1024 个多边形的存储空间。下面是一些可用于加载的.COB 物体：

- SPHERE01.COB（互换 YZ 坐标，反转环绕顺序）；
- CAR01.COB（互换 YZ 坐标，执行局部变换和世界变换）；
- HAMMER03.COB（互换 YZ 坐标，执行局部变换和世界变换）。

警告：创建.COB 文件时，一定要将物体划分为三角形，同时加载函数只能读取一个物体。另外，一定要将每个多边形的颜色设置为“plain color”，将反射模型设置为“matte”，以便执行恒定着色（当前我们的引擎只支持这种着色方法）。

8.5.5 Quake II 二进制.MD2 格式概述

Quake II .MD2 格式是最流行的 3D 动画格式之一。这种格式是为 Quake II（即游戏）设计的，诸如 Max 和 trueSpace 等建模程序不支持它。有用于创建和查看这种格式模型的工具，这将在以后介绍；当前，读者只需对这种格式支持哪些信息有大概的了解。.MD2 文件表示一个 Quake II 游戏模型，包含一组顶点、动画帧、皮肤（skin）等信息。每个.MD2 文件都是一个完整的 Quake II 角色动画集，包含该角色的每种动作（行走、死亡、射击、转身等）的动画序列，总共 384 帧。



图 8.51 一个 Quake II .MD2 格式的皮肤

皮肤是方形纹理图，通常是.PCX 文件格式，大小为 256×256，被用作模型的纹理。所有的纹理都包含在同一个皮肤中，通过使用纹理坐标从中切取所需的纹理。图 8.51 和 8.52 是一个应用于.MD2 网格的皮肤。

图 8.52 是一个.MD2 模型的实际网格（使用 Milkshape 3D 查看）。这个模型是从 http://www.planetquake.com/polycount 下载的，这是一个很不错的提供模型下载的地方。当然，如果要将这里提供的模型用于商业游戏中，必须得到许可。

.MD2 格式还支持简单的压缩方案，如偏移（offset bias）、浮点数的 delta 编码等。总之，这是一种易于分析和理解的格式。当然，它是二进制的，但很不错。本书后面介绍动画时将使用这种格式。这种格式惟一的缺点是不支持材质，因此我们加载这种格式的物体时，必须应用默认材质，让光照引擎知道如何处理模型。

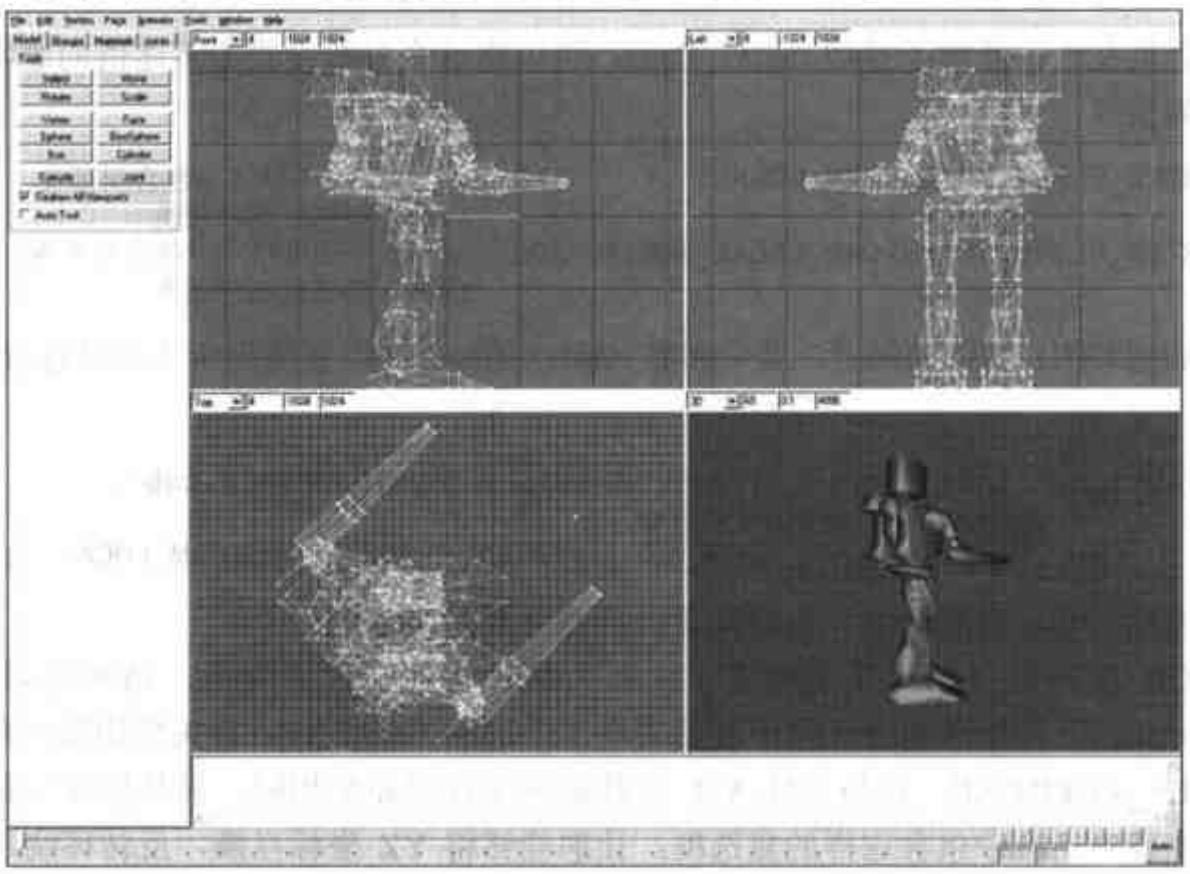


图 8.52 Quake II .MD2 网格中的一帧

8.6　3D 建模工具简介

在本章结束之前，简要地介绍一些对于建模和转换工作很有帮助的工具（附带光盘中包含其中一些工具的评估版）：

- **Caligari trueSpace 4.0+**（建模程序）——如图 8.53 所示，这是作者最喜欢的建模程序之一。它速度快，易于使用，价格合理，能够以很多文件格式导出模型。更详细的信息请访问 http://www.caligari.com。

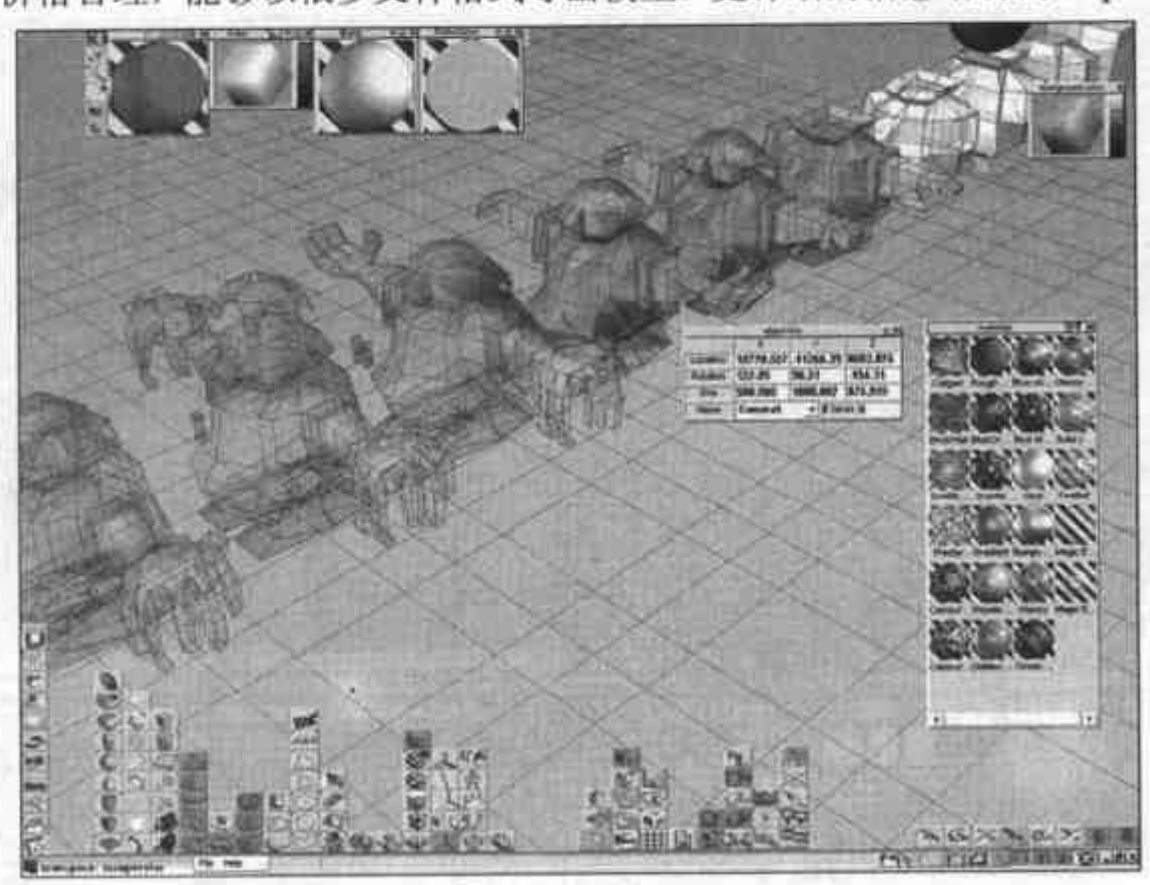


图 8.53　trueSpace

- **3D Studio Max 3.0+**（建模程序）——如图 8.54 所示，简而言之，Max 非常棒，可用于制作游戏、电影和任何您能想到的东西。不过，2500～5000 美元的价格可能会让您望而却步。更详细的信息请访问 http://www.discreet.com/products/3dsmax/。

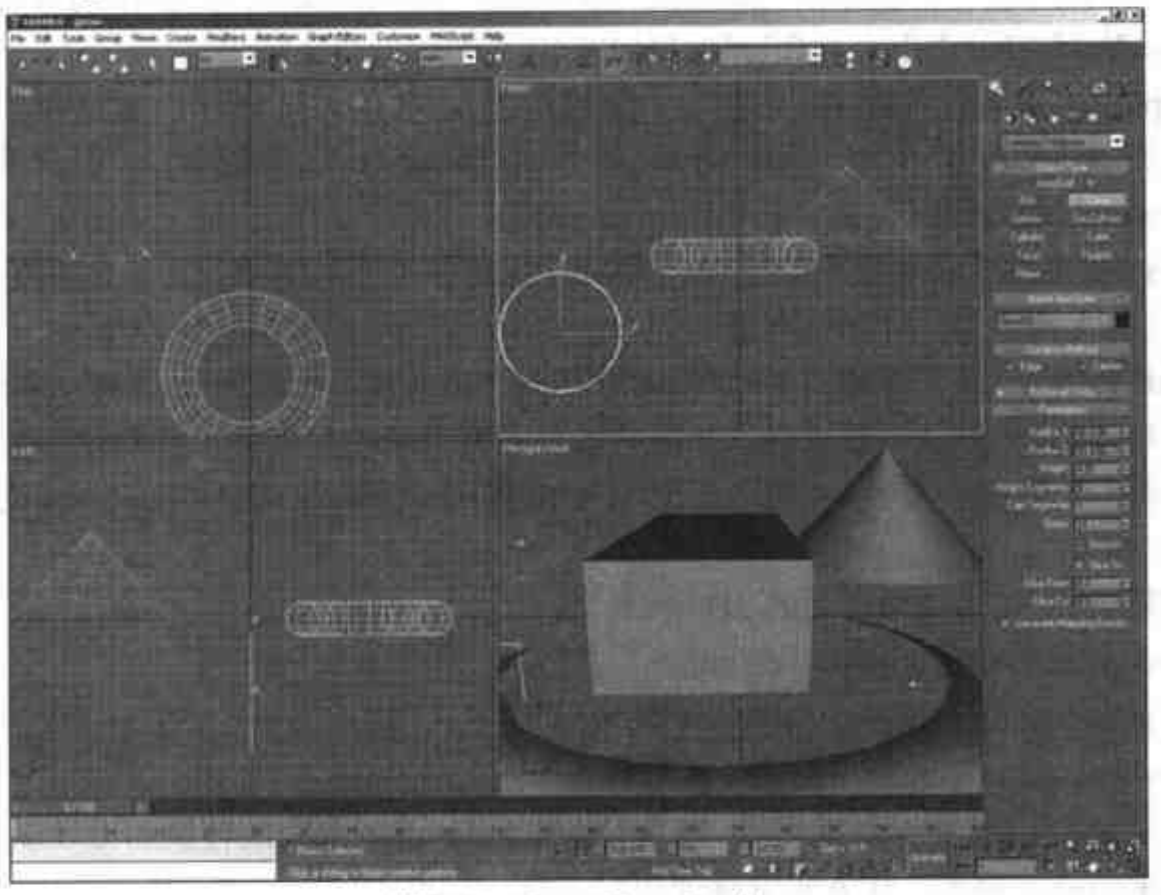


图 8.54　Gmax Game Studio

● **Gmax**——类似于 3D Studio Max，如图 8.54 所示。这是一个有趣的产品。从表面上看，它是免费的，但实际上是 3D Studio Max 的简化版。作者对其了解不多，但绝对值得购买。您可以以游戏开发人员的身份获得使用许可，更详细的信息请访问 http://www.discreet.com/products/gmax/。

● **Blender**（建模程序）：如图 8.55 所示，这是一个免费建模程序，但公司 NaN 处于暂时歇业状态，在作者看来，“产品免费发送，只收取技术支持费”的运营理念是行不通的。这个建模程序很是了得，真心希望设计厂商能将其转让给他人，或通过重新包装后作为产品出售。更详细的信息请访问 http://www.blender.nl。

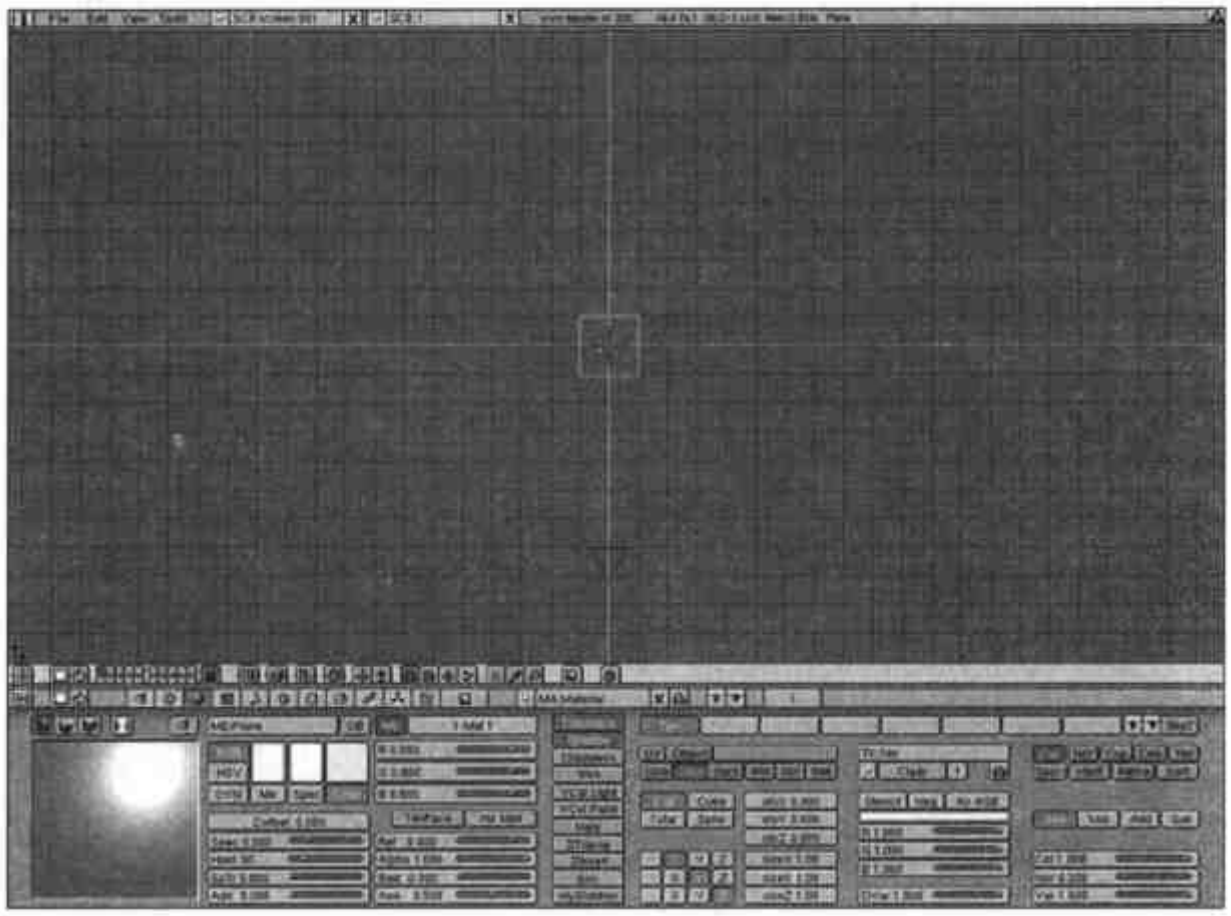


图 8.55 Blender

● **Milkshape** 3D（建模程序、查看程序和导出器）：如图 8.52 所示，这是一个功能非常强大的工具，是专门针对游戏文件格式设计的。它是一个开放源代码工具，很多人在对其进行改进，这有好处也有坏处。好处在于它几乎是免费的，坏处在于没有专门介绍该产品的网站。一个长期介绍给软件的网站是 http://www.swissquake.ch/chumbalum-soft/index.html。

● **Deep Exploration**（查看程序、导出器和建模程序）：如图 8.56 所示，这是作者见到的最强大的工具，其界面可谓鬼斧神工，让用户能够在各种模式下查看数据。更详细的信息请访问 http://www.righthemisphere.com。

● **Polytrans**（查看程序和导出器）：它是功能最完备的导入/导出系统之一，几乎支持每一种 3D 格式，但对游戏文件格式的支持不强。详细信息请访问 http://www.okino.com/conv/conv.htm。

● **WorlCraft Editor**（关卡编辑器）：如图 8.57 所示，它是数不胜数的 mod 和众多游戏（如 Half-Life）的幕后功臣。可以使用它来创建游戏世界，然后将其导入到引擎中。当前，我们并不需要这样的工具，但创建室内游戏时将需要使用它。有很多致力于介绍 WorldCraft 的网站，其中 http://www.valve-erc.com/一直在从事这方面的工作。

下面是一些很不错的提供模型资源的 3D 网站：

- http://www.planetquake.com/polycount/；
- http://www.web3d.org/vrml/vft.htm；
- http://www.gmi.edu/~jsinger/free3dmodels.htm。

通常，作者使用 trueSpace 和 Max 来建模，使用 Deep Exploration 来完成转换和处理工作。不过，Milkshape 3D 的功能比看起来的强大得多，作者将继续使用它。

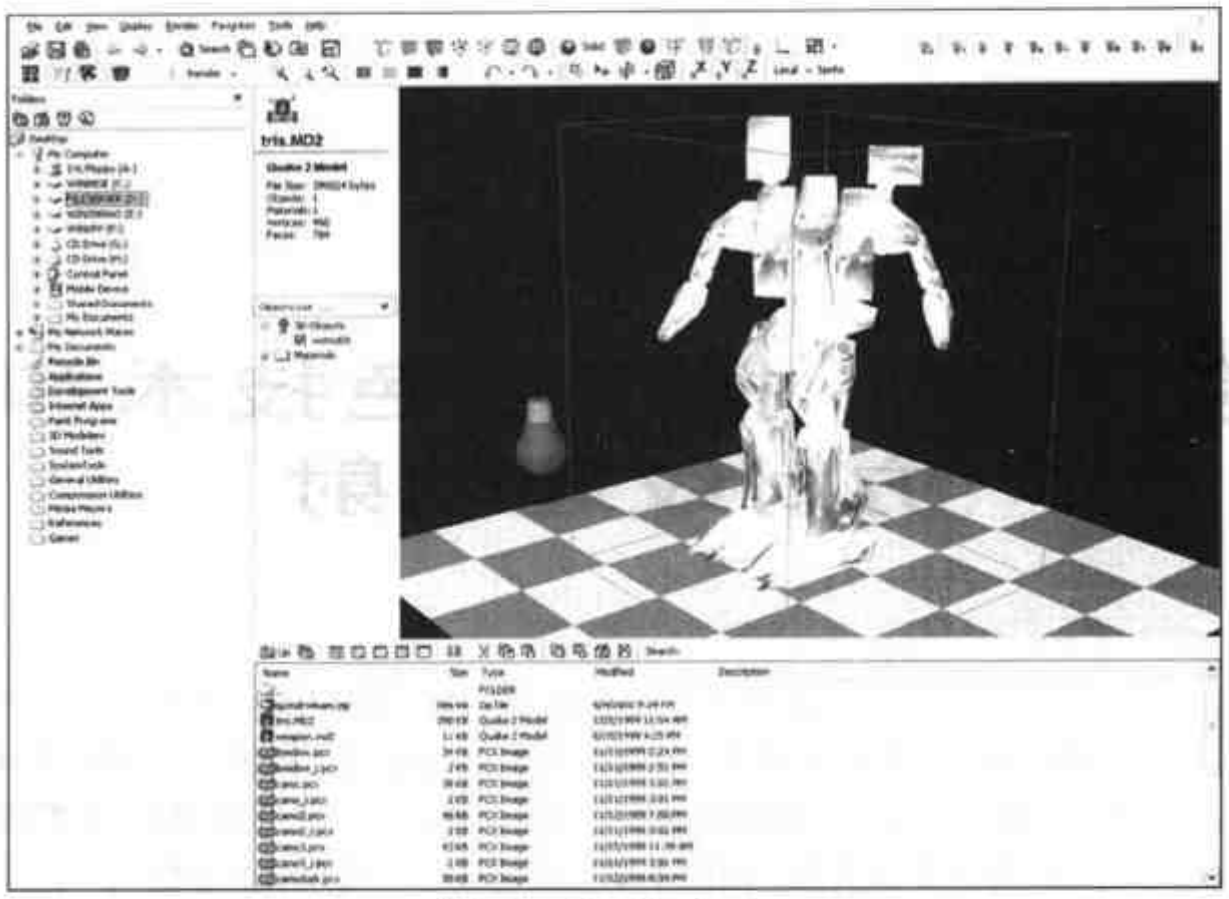


图 8.56　Deep Exploration

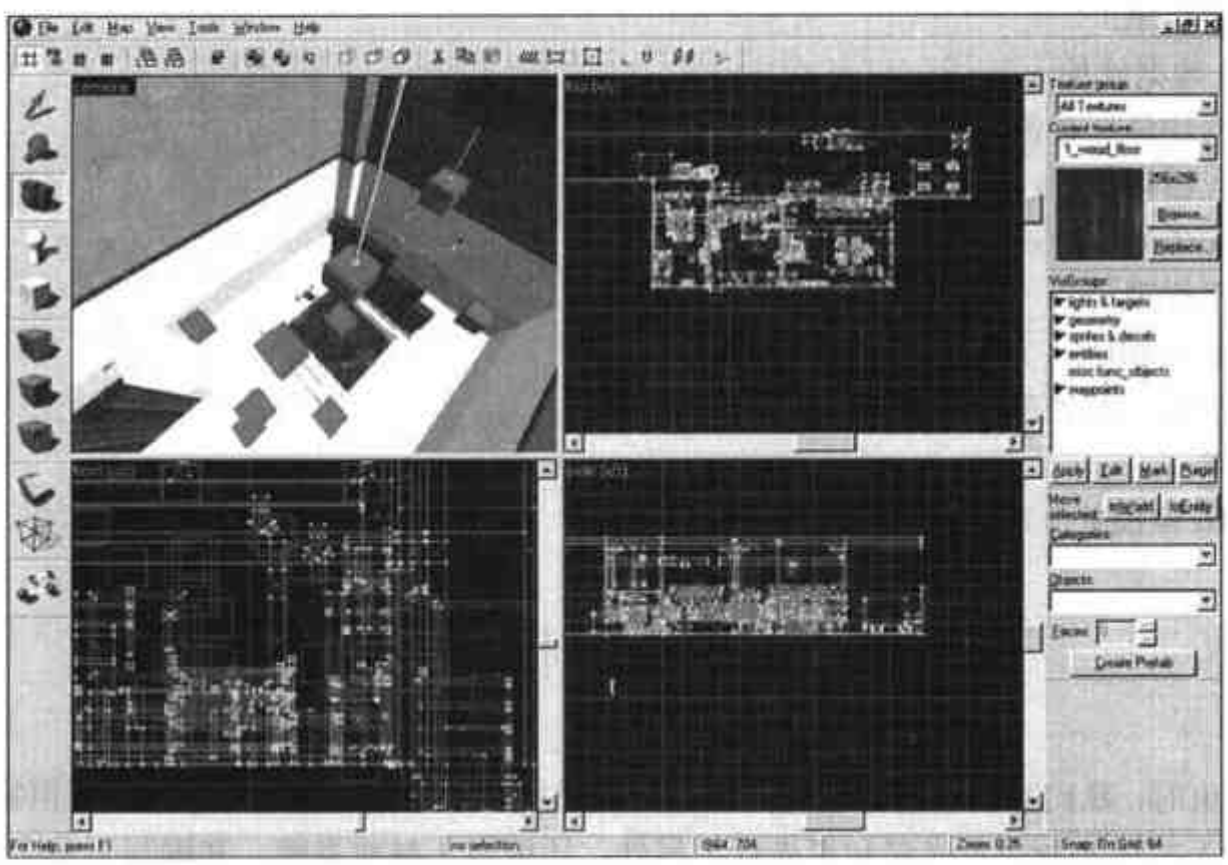


图 8.57　WorldCraft Editor

8.7　总　　结

本章的内容很棘手，涉及到理论和实践。本章介绍的内容很多，包括基本光照理论、材质、光源、大量的数学公式、着色、深度排序和新的模型格式等。读者的头脑中可能充斥了各种想法，后面将向读者提供实现这些想法的机会。接下来的一章包含的内容少得多，将讨论像素级着色、纹理映射，并详细介绍三角形的光栅化。到下一章结束时，我们的 3D 引擎将全面支持纹理映射和光照（当然，这是一种室外游戏引擎）。

第 9 章　插值着色技术和仿射纹理映射

作者刚完成本章的编码工作。仅核心引擎需要修改和添加的代码就超过 15 000 行，另外演示程序的代码大约有 5000 行，这应该能够满足本章的要求。作者的目的不是给读者提供一个可行的 3D 引擎，然后说“祝您玩得愉快”，而是要引导读者循序渐进地创建多个 3D 引擎，以阐述创建 3D 引擎的思路。

作者对 16 位核心引擎的性能非常满意，因此本章可能是最后一次介绍 8 位模式，以后将不再需要它。到进行这种过渡时，作者将阐述具体的理由。本章将介绍的内容如下：

- 讨论新的引擎；
- 更新 3D 数据结构；
- 重新编写物体加载函数；
- 多边形光栅化回顾；
- 实现 Gouraud 着色；
- 基本的采样理论；
- 实现仿射（affine）纹理映射；
- 对纹理进行光照处理；
- 函数清单；
- 8 位和 16 位模式下的优化策略。

9.1　新 T3D 引擎的特性

在上一章结束时，我们的引擎能够使用光照系统来渲染实心物体，该光照系统采用恒定着色光照模型，支持环境光源、无穷远光源、点光源和聚光灯。另外，还编写了材质系统，并编写了几个文件加载函数和支持函数，用于加载 3D Studio .ASC 和 Caligari .COB 文件。只有 Gouraud 着色和基本纹理映射没有处理，这将在本章完成。

如果读者认为这两项工作没有什么大不了的，您就错了。原因在于编写纹理映射模块和 Gouraud 着色模块本身并不难，难的是以有效的方式将其集成到引擎中。编写一个这样的程序并不难：在相机前面旋转 3D 立方体，并使用特定的光源和纹理对其进行处理；难的是编写一个支持任何相机、光源、模型、纹理、分辨率、位深、文件格式等通用的系统。这正是我们一直在追求的目标。集成 Gouraud 着色和纹理映射是一个巨大的挑战，需要重新编写引擎，在数据结构、光照模块、模型加载函数中考虑一些棘手的细节。另外，还需要进行优化，例如，预先计算法线的长度以方便对多边形执行恒定着色处理，将每个光源的数学运算结果存储到高速缓存中以提高速度。

虽然前一章花了很大的篇幅讨论材质和材质数据库，但并没有为将完整的材质系统集成到引擎中做好充分准备。要支持纹理、纹理坐标和完整的 Gouraud 着色处理，必须对数据结构进行修改，提供很多数据结构的 2.0 版（这

将在稍后进行）。然而，这样做的一项副作用是，必须重新编写所有的函数——至少是那些涉及 3D 的函数。

当然，在很多情况下，只需修改函数原型和复制代码；但有些情况下，必须考虑数据结构中新增的元素。另外，还必须对光照系统的设计和颜色信息流程进行修改，即为每个多边形提供用于存储光照处理后的颜色的空间。

此时速度问题将显现出来。在需要优化代码而又无法使用易于理解的代码来实现优化时，作者将讨论这一点。换句话说，在某些地方（如纹理映射模块中），作者不得不使用低级（low-level）技巧来提供性能，当然不是使用汇编语言，而是使用 SIMD（单指令多数据）指令来执行并行处理，这样至少可以将速度提高 4 倍。有关这方面的内容将在第 16 章介绍。

总之，新的引擎版本支持在 8 位和 16 位模式下对单色多边形进行 Gouraud 着色和光照处理以及对纹理映射多边形进行恒定着色。下面来讨论一些新的引擎软件。这不会太长，因为作者不喜欢在不涉及多少理论的情况下堆砌大量的代码。不过话又说回来，如果不知道自己要做什么，理论就没有多大帮助，因此这里先介绍一下新引擎。

警告：2.0 版引擎假设在 16 位模式下 RGB 颜色格式为 5.6.5。之所以这样做是因为动态地判断 RGB 颜色格式会降低性能。可以使用条件编译来支持两种 RGB 颜色格式，但坦率地说，当前几乎没有使用 5.5.5 格式的图形卡，因此这样做没有意义。

9.2　更新 T3D 数据结构和设计

首先，新的库模块如下：

- T3DLIB7.CPP——包含核心 C/C++函数；
- T3DDLIB7.H——头文件。

和往常一样，编译本章的程序时，必须链接所有的库模块：T3DLIB1.CPP～T3DLIB7.CPP。正如前面指出的，本章旨在添加 Gouraud 着色和仿射纹理映射功能（仿射纹理映射是线性的，不涉及透视修正）。在本章中，作者面临的困境是，虽然从表面上看，实现 Gouraud 着色和仿射纹理映射很简单，但通过深入分析将发现，为支持新的光照和纹理映射特性，必须对数据结构做重大修改。这样做的副作用是，必须重新创建所有的函数。

提示：C++拥趸可能会说，使用 C++时这没有什么大不了的！然而，这确实是个问题；是的，只需派生出另一个类，然后在其中实现新增的内容；但问题正是必须派生出另一个类。作者不想实现类层次，而只想使用单组数据结构和函数。从这种意义上说，我们创建该引擎的方式是最为高效的。

令人欣慰的是，在大多数情况下，只需对前一版引擎中相应函数的几行关键代码进行修改即可，当然还需要创建新的函数原型。虽然本章将介绍所有新增的内容，但为避免读者来回翻阅，这里做一下预览，看看新头文件 T3DLIB7.H 中的重要元素及其在引擎中的用途。

警告：读者运行 8 位演示程序时将发现，由于需要计算查找表，可能出现长达 1 分钟的延迟，在此期间读者看到的是黑屏，请不要恐慌。

9.2.1　新的#defines

这里列出用于新的库模块 T3DLIB7.CPP 的#defines，它们都位于头文件 T3DLIB7.H 中。这里按用途将它们分类。下面是一些帮助纹理映射函数绘制三角形的常量：

```
// 为帮助纹理映射函数分析三角形定义的常量
#define TRI_TYPE_NONE          0
#define TRI_TYPE_FLAT_TOP      1
#define TRI_TYPE_FLAT_BOTTOM   2
#define TRI_TYPE_FLAT_MASK     3
#define TRI_TYPE_GENERAL       4
#define INTERP_LHS             0
#define INTERP_RHS             1
#define MAX_VERTICES_PER_POLY  6
```

上述常量被纹理映射函数的状态机用来帮助绘制三角形。读者可以不用关心这些常量，但有趣的是，作者让纹理映射函数渲染常规三角形，而不是将三角形分割成平底三角形和平顶三角形，这些常量可帮助完成这种工作。

接下来的#defines 用于新的多边形结构 POLY4DV2，读者对它们应有似曾相识之感：

```
// 为第 2 版多边形和多边形面结构定义的常量

// 多边形和多边形面的属性
#define POLY4DV2_ATTR_2SIDED        0x0001
#define POLY4DV2_ATTR_TRANSPARENT      0x0002
#define POLY4DV2_ATTR_8BITCOLOR      0x0004
#define POLY4DV2_ATTR_RGB16         0x0008
#define POLY4DV2_ATTR_RGB24         0x0010

#define POLY4DV2_ATTR_SHADE_MODE_PURE      0x0020
#define POLY4DV2_ATTR_SHADE_MODE_CONSTANT    0x0020 // 别名
#define POLY4DV2_ATTR_SHADE_MODE_EMISSIVE    0x0020 // 别名

#define POLY4DV2_ATTR_SHADE_MODE_FLAT      0x0040
#define POLY4DV2_ATTR_SHADE_MODE_GOURAUD    0x0080
#define POLY4DV2_ATTR_SHADE_MODE_PHONG     0x0100
#define POLY4DV2_ATTR_SHADE_MODE_FASTPHONG   0x0100 // 别名
#define POLY4DV2_ATTR_SHADE_MODE_TEXTURE     0x0200

// 新增的
#define POLY4DV2_ATTR_ENABLE_MATERIAL 0x0800 // 根据材质进行光照计算
#define POLY4DV2_ATTR_DISABLE_MATERIAL 0x1000 // 根据多边形颜色进行光照
                    //计算（就像以前那样）

// 多边形和多边形面的状态
#define POLY4DV2_STATE_NULL         0x0000
#define POLY4DV2_STATE_ACTIVE        0x0001
#define POLY4DV2_STATE_CLIPPED       0x0002
#define POLY4DV2_STATE_BACKFACE       0x0004
#define POLY4DV2_STATE_LIT          0x0008
```

如果读者仔细地阅读过前一个引擎的代码，应该会发现所有的 POLY4DV2_*常量都与以前的 POLY4DV1_*常量相同——只有几个常量是新增的，它们为粗体。新的#define 让数据结构能够支持材质以及指出多边形是否经过光照处理。之所以需要最后一个常量，是因为现在有多条光照流水线路径，通过使用这个常量，可避免重复对多边形进行光照处理。

接下来的#defines 定义了顶点控制标记，在加载物体时，它们被作为参数传递给物体加载函数。这些标记用于帮助覆盖某些属性（如着色模型）以及提供某种信息（如是否要反转纹理坐标）。

```
// (新增的)用于简单模型格式，指定着色模式
#define VERTEX_FLAGS_OVERRIDE_MASK          0xf000 // 该掩码用于提取值

#define VERTEX_FLAGS_OVERRIDE_CONSTANT 0x1000
#define VERTEX_FLAGS_OVERRIDE_EMISSIVE 0x1000 //别名
#define VERTEX_FLAGS_OVERRIDE_PURE    0x1000
#define VERTEX_FLAGS_OVERRIDE_FLAT    0x2000
#define VERTEX_FLAGS_OVERRIDE_GOURAUD 0x4000
#define VERTEX_FLAGS_OVERRIDE_TEXTURE 0x8000

#define VERTEX_FLAGS_INVERT_TEXTURE_U 0x0080   // 反转纹理坐标 u
#define VERTEX_FLAGS_INVERT_TEXTURE_V 0x0100   // 反转纹理坐标 v
#define VERTEX_FLAGS_INVERT_SWAP_UV    0x0800   // 将纹理坐标 u 和 v 互换
```

后面讨论物体加载函数时，将详细讨论这些标记。接下来的#define 与物体数据结构 OBJECT4DV2 相关，其中大多数常量的值与以前相同，只是名称被改为 OBJECT4DV2_*。有 3 个常量是新增的，它们被显示为粗体。

```
// 为第 2 版物体结构定义的常量
// 现在物体采用动态内存分配方式，但仍指定了最大顶点数和多边形数
#define OBJECT4DV2_MAX_VERTICES    4096
```

```
#define OBJECT4DV2_MAX_POLYS       8192

// 物体的状态
#define OBJECT4DV2_STATE_NULL      0x0000
#define OBJECT4DV2_STATE_ACTIVE    0x0001
#define OBJECT4DV2_STATE_VISIBLE   0x0002
#define OBJECT4DV2_STATE_CULLED    0x0004

// 新增的
#define OBJECT4DV2_ATTR_SINGLE_FRAME      0x0001 // 单帧物体（和以前一样）
#define OBJECT4DV2_ATTR_MULTI_FRAME       0x0002 // 多帧物体，用于支持.md2
#define OBJECT4DV2_ATTR_TEXTURES          0x0004 // 指出物体是否包含
                          //带纹理的多边形?
```

OBJECT4DV2_ATTR_SINGLE_FRAME 和 OBJECT4DV2_ATTR_MULTI_FRAME 用于支持包含多帧的物体。我们想实现类似于 Quake II .MD2 格式的“网格动画”，为此，需要使用某种方式对物体进行标记，指出其顶点列表中是否包含多个帧。选择当前帧后，只使用该帧中包含的顶点，虽然顶点与多边形的对应关系、多边形颜色等不变，这将在后面详细介绍。最后，属性 OBJECT4DV2_ATTR_TEXTURES 指出物体的某些多边形被贴上纹理，这可帮助模型进行优化。

接下来的常量指定渲染列表中最多可包含多少个多边形。这可以是任何值，读者可以修改它。不过，作者认为，要达到 15～30 帧/秒的速度，每帧中包含的多边形不能超过 10 000 个，因此将该常量设置为 32 768 足够了。

```
// 为第 2 版渲染列表结构定义的常量
#define RENDERLIST4DV2_MAX_POLYS     32768
```

下面的#defines 很有趣（其中的注释说明了这一点）。从现在开始，将使用更健壮的顶点结构来定义顶点，这种结构不但应包含顶点坐标，还应包含法线、纹理坐标和属性标记。

```
// 为顶点数据结构定义的常量
// 给变换系统和光照系统以提示，以便判断顶点是否有有效的法线（需要旋转）
// 或者是否有纹理坐标（需要进行裁剪）
// 从而最大限度地减少光照处理和渲染的工作量
// 因为通过这些属性可以知道处理的顶点的格式
// 这种格式类似于 Direct3D 的灵活顶点格式，可以存储:
// 3D 点
// 3D 点 + 法线
// 3D 点 + 法线 + 纹理坐标
#define VERTEX4DTV1_ATTR_NULL      0x0000 // 顶点为空
#define VERTEX4DTV1_ATTR_POINT     0x0001
#define VERTEX4DTV1_ATTR_NORMAL    0x0002
#define VERTEX4DTV1_ATTR_TEXTURE   0x0004
```

作者对新的顶点数据结构 VERTEX4DV1 很满意，它简化了很多工作，将在稍后讨论数据结构时介绍它。

至此，所有新增的#defines 都介绍过了，这并不多。以循序渐进的方式进行学习至关重要，这样将在学习的过程中获得丰富的经验。别忘了，知道不该做什么与知道该做什么同样重要。

9.2.2　新增的数学结构

如果读者查看本书的最新数学库，将发现其中的内容很多。在这方面我们可能走得太远，实际上每章需要的新的数学功能很少，其中只有一些名称更清晰的数据结构是必不可少的。本章需要的是一些整型向量类型，如下所示：

```
// 整型 2D 向量，不带 W 坐标点
typedef struct VECTOR2DI_TYP
{
union
  {
  int M[2]; // 以数组方式存储

  // 以名称存取
  struct
     {
```

```
      int x,y;
      }; // end struct

  }; // end union

} VECTOR2DI, POINT2DI, *VECTOR2DI_PTR, *POINT2DI_PTR;

// 整型 3D 向量、不带 w 坐标点
typedef struct VECTOR3DI_TYP
{
union
  {
  int M[3]; // 以数组方式存储

  // 使用名称存取
  struct
     {
     int x,y,z;
     }; // end struct

  }; // end union

} VECTOR3DI, POINT3DI, *VECTOR3DI_PTR, *POINT3DI_PTR;

// 整型 4D 齐次向量、包含 w 坐标的点
typedef struct VECTOR4DI_TYP
{
union
  {
  int M[4]; // 以数组方式存储

  // 通过名称存取
  struct
     {
     int x,y,z,w;
     }; // end struct
  }; // end union

} VECTOR4DI, POINT4DI, *VECTOR4DI_PTR, *POINT4DI_PTR;
```

它们只不过是 2D、3D 和 4D 整型向量，用于帮助执行纹理映射或向量格式存储数据。

9.2.3 实用宏

下面是一个不错的浮点数比较函数：

```
// 对浮点数进行比较
#define FCMP(a,b) ( (fabs(a-b) < EPSILON_E3) ? 1 : 0)
```

使用下面的代码来比较浮点数是一种糟糕的做法：

```
float f1, f2;

// 将 f1 和 f2 设置为相同的值

if (f1==f2)
  {
  // 执行逻辑
  }
```

问题存在于 if 条件语句中。虽然在您看来 $f1$ 和 $f2$ 可能相等，但由于四舍五入或精度不够，$f1$ 和 $f2$ 可能存在一些数字“噪声”，这样虽然从数学上说它们相等，但在比较时情况可能并非如此。因此，一种更好的做法是计算它们的绝对差，并将其同一个接近于 0 的值（如 0.001 和 0.0001）进行比较。实际上，这两个数都是 0，因为浮点数最多只精确到小数点后的 4～5 位。这就是为何使用 FCMP()宏的原因。

接下来的内联宏是复制函数，用于复制对象 VERTEX4DV1。从技术上说，这种对象并不需要复制运

算符，但作者想在优化之前将问题说清楚。

```
inline void VERTEX4DTV1_COPY(VERTEX4DTV1_PTR vdst, VERTEX4DTV1_PTR vsrc)
{ *vdst = *vsrc; }

inline void VERTEX4DTV1_INIT(VERTEX4DTV1_PTR vdst, VERTEX4DTV1_PTR vsrc)
{ *vdst = *vsrc; }
```

最后一个内联函数用于计算向量长度，其速度比以前的版本要快。惟一的不同之处在于，原来的 VECTOR4D_Length_Fast()通过调用函数来计算 3D 距离，这增加了函数调用开销。下面是改进后的版本：

```
inline float VECTOR4D_Length_Fast2(VECTOR4D_PTR va)
{
// 这个函数计算原点到点(x, y, z)的距离
int temp; // 用于交换
int x,y,z; // 用于计算距离

// 确保所有的值皆为正
x = fabs(va->x) * 1024;
y = fabs(va->y) * 1024;
z = fabs(va->z) * 1024;

// 排序
if (y < x) SWAP(x,y,temp)
if (z < y) SWAP(y,z,temp)
if (y < x) SWAP(x,y,temp)

int dist = (z + 11 * (y >> 5) + (x >> 2) );

// 计算的距离误差小于 8%
return((float)(dist >> 10));

} // end VECTOR4D_Length_Fast2
```

9.2.4　添加表示 3D 网格数据的特性

更新 3D 数据结构可能是最费劲的。问题在于直接给出最终的结果是毫无意义的，因此作者循序渐进地介绍每个版本，在每个版本中只添加完成工作所需的东西。但这样做的工作量很大。

在前一章中，我们实现了一个光照系统，它支持对每个多边形进行固定着色和恒定着色。文件格式 PLG/PLX 和.COB 都支持指定多边形的属性，而.ASC 格式只支持顶点、多边形和颜色。PLG/PLX 格式让我们能够使用下述标记手工设置多边形的着色方法：

```
// 用于改进 PLG 文件格式，以支持 PLX 的常量
// 面描述符仍为 16 位，但格式如下：
// d15             d0
//  CSSD | RRRR| GGGG | BBBB

// C 是 RGB/索引颜色标记
// SS 指定着色模式
// D 是双面标记
// 在 RGB 颜色模式下，RRRR、GGGG 和 BBBB 是红、绿、蓝分量
// 在 8 位模式下，GGGGBBBB 为 8 位的颜色索引
// 用于简化测试的掩码？
#define PLX_RGB_MASK     0x8000 // 用于提取 RGB/索引颜色模式的掩码
#define PLX_SHADE_MODE_MASK 0x6000 // 用于提取着色模式的掩码
#define PLX_2SIDED_MASK    0x1000 // 用于提取双面标记的掩码
#define PLX_COLOR_MASK 0x0fff // xxxxrrrrggggbbbb，每个 RGB 分量 4 位
                // xxxxxxxxiiiiiiii，8 位颜色索引

// 用于在提取多边形的颜色模式下检测其值
#define PLX_COLOR_MODE_RGB_FLAG   0x8000   // 多边形使用的是 RGB 颜色
#define PLX_COLOR_MODE_INDEXED_FLAG 0x0000  // 多边形使用的是 8 位颜色索引

// 双面标记
#define PLX_2SIDED_FLAG        0x1000  // 多边形是双面的
```

```
#define PLX_1SIDED_FLAG        0x0000  // 多边形是单面的

// 多边形的着色模式
#define PLX_SHADE_MODE_PURE_FLAG    0x0000 // 固定着色
#define PLX_SHADE_MODE_CONSTANT_FLAG 0x0000 // 别名
#define PLX_SHADE_MODE_FLAT_FLAG    0x2000 // 恒定着色
#define PLX_SHADE_MODE_GOURAUD_FLAG  0x4000 // Gouraud 着色
#define PLX_SHADE_MODE_PHONG_FLAG    0x6000 // Phong 着色
#define PLX_SHADE_MODE_FASTPHONG_FLAG  0x6000 // Phong 着色
```

对于.COB 模型，我们根据“Shader class: color”后面一行中的内容来确定多边形是着色的（colored）还是纹理映射的，根据“Shader class: reflectance”后面一行的内容来确定着色方法。.COB 物体加载函数中的注释说明了这种规则：

```
// 现在需要知道着色模型
// 为此，首先需要找到 "Shader class: color", then after this line is:
// 然后找到其后面的 "Shader name: "xxxxxx" (xxxxxx)"
// 对于不带纹理的多边形，xxxxx 为"plain color"和"plain"
// 对于带纹理的多边形，为"texture map"和"caligari texture"
// 然后找到"Shader class: reflectance"
// 并查看它后面的"Shader name: "xxxxxx" (xxxxxx)"
// 然后根据 xxxxxx 的值确定着色模式：
// "constant" -> MATV1_ATTR_SHADE_MODE_CONSTANT
// "matte"  -> MATV1_ATTR_SHADE_MODE_FLAT
// "plastic" -> MATV1_ATTR_SHADE_MODE_GOURAUD
```

然而，还存在一个光照计算方面的小问题。在多边形结构中，并没有提供存储光照处理后多边形最终颜色的空间。换句话说，多边形的初始颜色以 8/16 位的格式存储在多边形结构的 color 字段中，在光照计算中，根据该颜色和光源计算多边形的最终颜色。对渲染列表执行光照计算时，这不是什么大问题，因为可以对渲染列表中的任何多边形进行修改。然而，如果对还没有被插入到渲染列表中的物体进行光照计算，将遇到麻烦。

原因在于，不能用光照处理后的颜色覆盖多边形原来的颜色。为解决这种问题，我们采取的策略是，将光照处理后的颜色数据存储到 color 字段的前 16 位中，如图 9.1 所示。

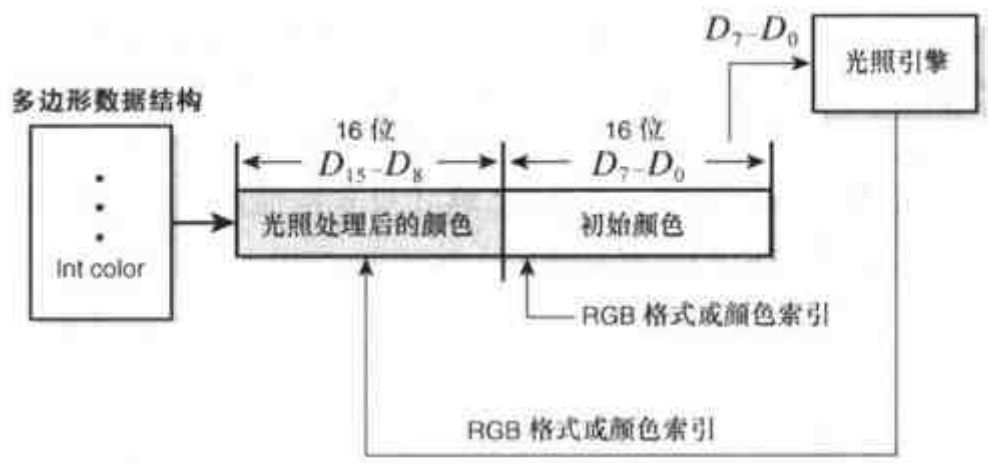


图 9.1　使用多边形结构中 color 字段的前 16 位来存储光照处理后的颜色

这种处理手法很管用，事实上，我们可以自豪地说：“真是一个了不起的想法”。然而，对多边形进行 Gouraud 着色时，这种方法不再管用。因此，需要在多边形数据结构中增加一个存储最终颜色的字段。

接下来需要考虑的是 Gouraud 着色算法本身。如图 9.2 所示，需要知道每个顶点处的法线（这是 Gouraud 着色的关键所在：计算每个顶点处的亮度，然后进行插值）。这要求我们考虑每个顶点的最终颜色，因此必须重新考虑如何为每个多边形/顶点提供光照处理后的颜色的空间。

最后，还需要增加对动画网格（如 Quake II .MD2 格式的动画网格）的支持。图 9.3 说明了这种概念，从本质上说，网格动画类似于标准的位图动画——是一系列被渲染的帧。假设有一个由 8 个顶点和 12 个多边形组成的立方体，其中每个多边形都是由顶点列表中的 3 个顶点定义的，有特定的颜色和着色模型。现在棘手的是如何添加帧。为此，我们创建 8 个新的顶点，它们的排列顺序与原来相同，但位置有细微的变化；然后用这些新顶点替换原来的顶点。这样多边形的定义、颜色和纹理保持不变，但物体在移动。

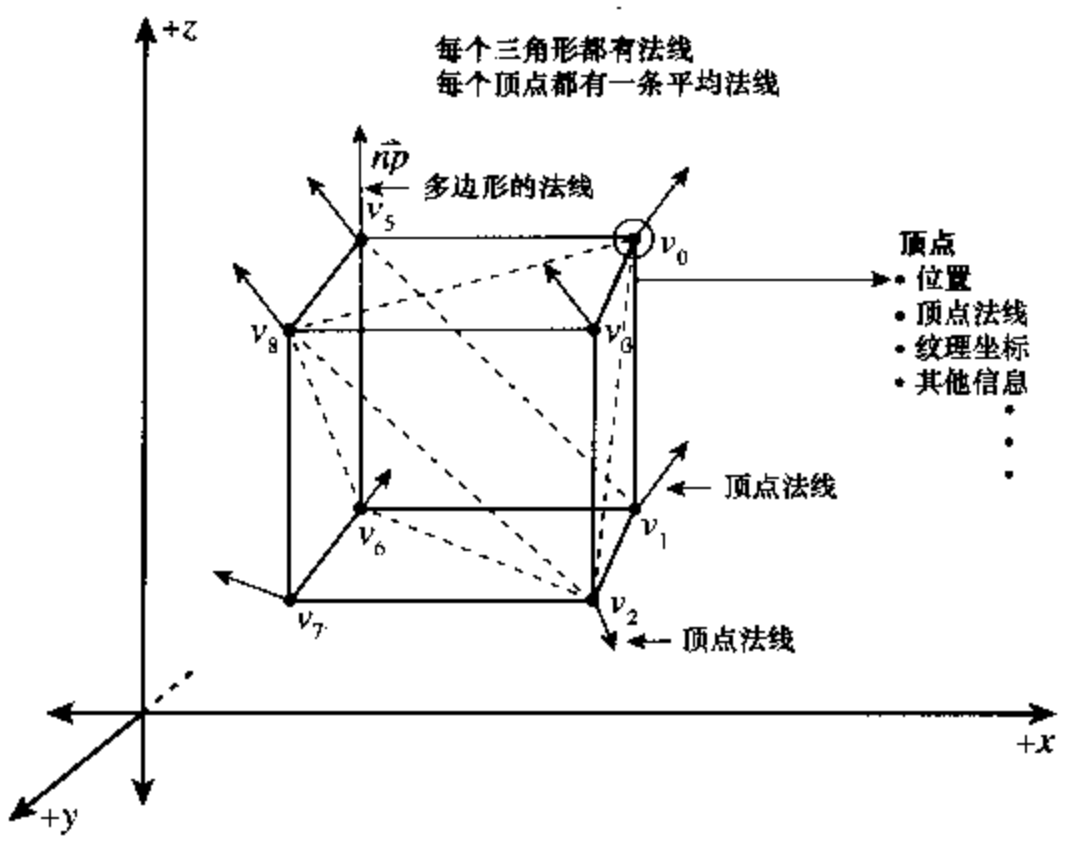


图 9.2　Gouraud 着色要求知道顶点处的法线

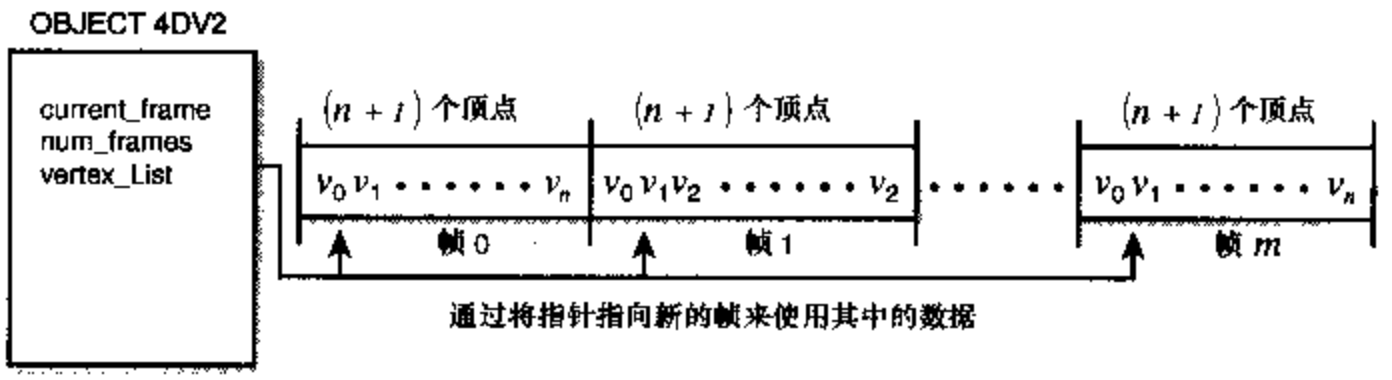


图 9.3　在物体中支持多个网格帧

Quake II .MD2 使用的就是这种技术。惟一的规则是，每个动画帧必须包含相同数量的顶点和多边形。顶点的位置可以变化，但它们与多边形的关系不变，即如果多边形 22 由顶点 12、90 和 200 组成，则在任何一帧中，该多边形都由这些顶点构成。

这里不打算创建.MD2 阅读器，因为本章包含的内容已经很多了。然而，在本书后面我们将支持这种文件格式的动画，同时多帧支持至关重要，因此在数据结构 OBJECT4DV2 中添加了这种功能，并在重新编写函数时考虑了这一点。

下面来看看新的顶点和多边形数据结构。

1. 顶点数据结构

到目前为止，我们还没有 3D 图形学意义上的“顶点”数据类型；在此之前我们的顶点数据类型是 3D 空间中的点，现在需要创建一些更强大的数据结构，以满足当前的需要。另外，还将使用前面定义的 VERTEX4DTV1_ATTR_*来灵活地控制顶点格式，以便能够创建不同类型的顶点，它们包含的信息不同，但结构和布局相同。下面是新的数据结构 VERTEX4DTV1：

注意：如果读者是 Direct3D 程序员，将知道为何灵活的顶点格式如此重要。如果没有这种灵活性，程序员必须使用预定义的顶点格式，这增加了编程工作的难度。

```
// 包含两个纹理坐标和顶点法线的 4D 齐次顶点
// 法线可被解释为向量或点
typedef struct VERTEX4DTV1_TYP
{
union
  {
    float M[12];     // 以数组方式存储
```

```
    // 通过名称存取
    struct
      {
      float x,y,z,w;    // 点
      float nx,ny,nz,nw; // 法线（向量或点）
      float u0,v0;      // 纹理坐标

      float i;       // 经过光照处理后的顶点颜色
      int   attr;    // 属性
      };         // end struct

     // 高级类型
     struct
       {
       POINT4D  v;    // 顶点
       VECTOR4D n;    // 法线
       POINT2D  t;    // 纹理坐标
       };

    }; // end union

} VERTEX4DTV1, *VERTEX4DTV1_PTR;
```

这是一个共用体，可以存储多种类型的数据，具体在其中如何存储数据，取决于您要如何编写算法。例如，如果想将顶点数据视为数据流（像 Direct3D 中那样），可以将数据存储为数组——M[ ]就是为此而设计的。同样，为让您能够使用名称来存取数据，其中提供了各种顶点数据的显式名称：

- x、y、z、w——顶点的坐标；
- nx、ny、nz、nw——顶点法线（稍后将详细介绍）；
- u0、v0——2D 纹理坐标；
- I——顶点的颜色或亮度；
- attr——顶点标记，指出了顶点的内容及其包含哪些信息。

另外，还可以以另一种结构方式存储，该结构中包含字段 v、u 和 t，它们分别是顶点坐标、法线和纹理坐标。要获得 3D 点的地址，可以使用下述代码：

```
VERTEX4DTV1 p1;

&p1.v
```

要获得 3D 点中 x 坐标的地址，可以这样做：

```
&p1.x
```

或

```
&p1.v.x
```

这样，可以将顶点坐标、法线或纹理坐标作为一个整体来传递，也可以访问各个分量。现在，创建顶点时，可以指定其法线（对光照计算来说至关重要）和纹理坐标，但仅当需要它们时才这样做。顶点类型是使用 attr 字段来指定的。例如，如果希望顶点支持 3D 点和法线，但不支持纹理坐标，可以这样设置 attr：

```
VERTEX4DTV1 p1;
p1.attr = VERTEX4DTV1_ATTR_POINT | VERTEX4DTV1_ATTR_NORMAL;
```

该字段只是提供了一项信息，让您和您的函数知道顶点中的哪些信息有效，并不能阻止您访问 3D 点、法线和纹理坐标。

2. 新的内部 PLOY4DV2 结构

下一个经过重大修改的数据结构是多边形结构。我们有两个这样的数据结构：一个引用了另一个结构（如物体）中的顶点列表，另一个完全是自包含的，用于构建渲染列表。如图 9.4 所示，其中的 POLY4DV2 引用外部顶点数据，其定义如下：

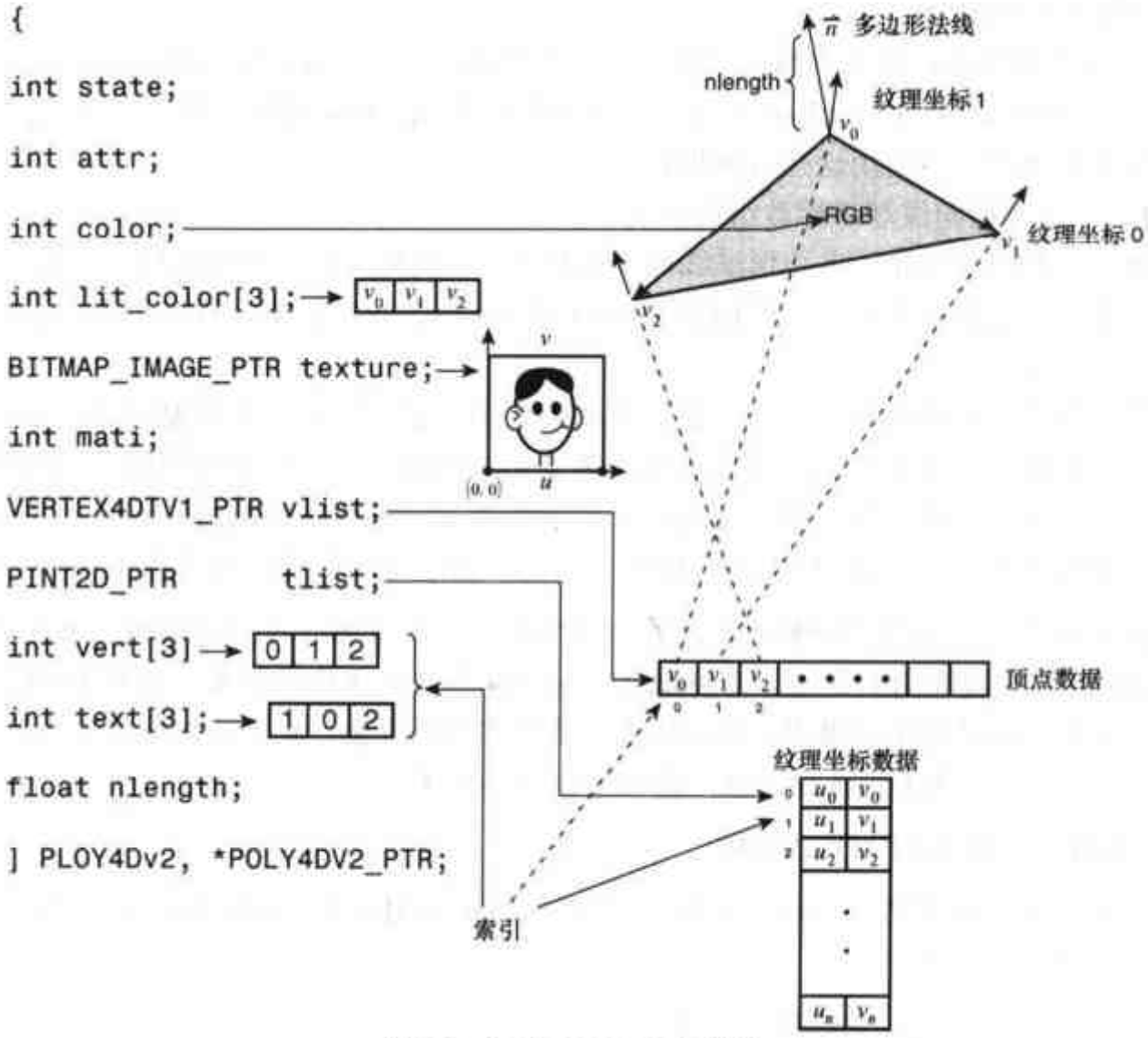


图 9.4　新的 POLY4DV2 结构

```
// 2.0版多边形结构，基于外部顶点列表
typedef struct POLY4DV2_TYP
{
int state;       // 状态信息
int attr;        // 物理属性
int color;       // 多边形的颜色
int lit_color[3];  // 用于存储经过光照处理后的颜色
          // 对于恒定着色，多边形颜色存储在第一个元素中
          // 对于Gouraud着色，顶点颜色分别存储在三个元素中

BITMAP_IMAGE_PTR texture;
             // 指向纹理信息的指针，用于简单纹理映射

int mati;        // 新增的，-1表示没有材质

VERTEX4DTV1_PTR vlist; // 顶点列表
POINT2D_PTR   tlist; // 新增的，纹理坐标列表
int vert[3];     // 指向顶点列表的索引
int text[3];     // 新增的，指向纹理坐标列表的索引
float nlength;   // 新增的，法线长度

} POLY4DV2, *POLY4DV2_PTR;
```

其中新增或修改过的字段为粗体。下面讨论其中每个字段的用途：

- **lit_colors[3]**——新的光照系统需要支持多边形颜色，在使用 Gouraud 着色时，还需要支持每个顶点的颜色，lit_colors[3]提供了这种特性。对采用恒定着色的多边形执行光照计算时，结果不再被存储到 color 的前 16 位中，而是被存储到 lit_color[0]中。采用 Gouraud 着色时，每个顶点的最终颜色分别被存储在 lit_color[0]、lit_color[1]和 lit_color[2]中。
- **texture**——出于简化的目的，现在仍没有提供全面的材质支持，但对于纹理映射多边形，需要提供存储纹理图的空间。texture 是一个指向纹理图的指针。

- **mati**——这是一个指向材质数据库的索引，供以后支持材质时使用。作者不知道是否会使用它，这里添加它旨在避免某些麻烦。
- **tlist**——这是新增的，用于支持纹理映射。如图 9.4 所示，当纹理图被映射到多边形上时，需要指定纹理坐标。新的 OBJECT4DV2 结构将包含一个纹理坐标数组，tlist 指向该数组，就像 vlist 指向顶点数组一样。在纹理映射阶段，将使用这些纹理坐标。
- **text[3]**——包含指向纹理坐标数组的索引。
- **nlength**——该字段存储了多边形法线向量的长度，以提高光照计算的速度。这是一个小小的优化，但让作者很伤脑筋。在光照引擎中，使用向量 v0->v1 和 v0->v2 的叉积来计算多边形的法线，然后对其进行归一化或计算其长度。

长度计算比点积更让作者担心。这不仅是因为我们使用的是近似值，更重要的是，根本没有必要计算长度，因为当多边形被旋转或平移后，法线长度不会变化。因此，只要总是使用两个相同的向量来计算，便可以预先计算所有多边形的法线长度。另外，在加载物体期间离线计算法线长度时，不必调用实时的长度近似函数（其速度更快），从而获得更高的精度。总之，通过这种优化，可节省不少的 CPU 周期。

提示：读者可能会问，为何不顺带预先计算多边形的法线向量呢？可以这样做，并对其进行归一化，但当物体被变换时，将需要旋转该向量，这将抵消预先计算多边形法线的好处。使用 4×4 矩阵进行旋转的计算量可能等于甚至超过叉积的计算量，因此这看似是个不错的注意，其实预先计算多边形法线毫无帮助，因为需要使用 4×4 矩阵来对法线进行变换，结果可能得不偿失。

3. 新的外部自包含 POLYF4DV2 结构

下一个经过重新修订的多边形结构是完全自包含的 POLYF4DV2，它用于在渲染列表中存储最终的多边形信息。这个结构的定义如下：

```
// 第 2 版的自包含多边形结构，供渲染列表使用
typedef struct POLYF4DV2_TYP
{
int state;        // 状态信息
int attr;        // 多边形的物理属性
int color;       // 多边形的颜色
int lit_color[3]; // 用于存储经过光照处理后的颜色
                  // 对于恒定着色，多边形颜色存储在第一个元素中
                 // 对于 Gouraud 着色，顶点颜色分别存储在三个元素中
BITMAP_IMAGE_PTR texture; // 指向纹理信息的指针，用于简单纹理映射

int    mati; // 新增的，-1 表示没有材质

float   nlength; // 新增的，法线长度
VECTOR4D normal; // 新增的，多边形法线

float    avg_z;   // 新增的，平均 z 值，用于简单排序

VERTEX4DTV1 vlist[3];  // 三角形的顶点
VERTEX4DTV1 tvlist[3]; // 变换后的顶点

POLYF4DV2_TYP *next;   // 指向下一个多边形的指针?
POLYF4DV2_TYP *prev;   // 指向前一个多边形的指针?

} POLYF4DV2, *POLYF4DV2_PTR;
```

POLYF4DV2 与 POLY4DV2 几乎完全相同。惟一不同的地方是，它在内部存储了顶点数据，而不是引用外部的顶点列表。另外，VERTEX4DTV1 表示的顶点包含纹理坐标，因此，需要的每个顶点的信息都包含在 vlist[ ]和 tvlist[ ]中。除此之外，其他字段的名称和含义都与 POLY4DV2 中相同。

9.2.5 更新物体结构和渲染列表结构

接下来需要更新的是最大的数据结构。在原来的设计中，我们使用通用的物体结构来表示物体，这种

结构中包含物体的顶点和多边形信息。对于在游戏世界中移动的物体而言，这种结构很管用，但对于诸如游戏关卡（game level）等更高级的数据，这种结构还管用吗？在大多数情况下，答案是肯定的。事实上，我们可能使用一系列的物体来表示一个关卡，其中每个物体表示一层楼或一个房间。当然，后面讨论空间划分时，将介绍为支持分层几何（level geometry）的概念需要对物体进行什么样的变换和修改。但就现在而言，这种物体结构是可行的，我们将继续使用。

新版本的物体结构 OBJECT4DV2 与原来的版本几乎相同，但支持纹理坐标、新的属性以及一个全局纹理（稍后将介绍）。然而，最大的不同是，新的 OBJECT4DV2 结构支持多网格动画（帧）。不过现在作者还没有编写任何使用它的代码（需要一个.MD2 阅读器）。图 9.3 说明了 OBJECT4DV2 以及每帧是如何存储的。该结构的定义如下，请将重点放在以粗体显示的字段上：

```
// 第 2 版物体结构，基于一个顶点列表和一个多边形列表
// 该结构有更大的灵活性，支持分帧动画
// 也就是说，它可以存储一个动画网格的数百帧
// 这些帧包含的多边形和几何体相同，但顶点位置不同
// 类似于 Quake II .md2 格式
typedef struct OBJECT4DV2_TYP
{
int  id;      // 物体的数字 ID
char name[64]; // 物体的名称
int  state;    // 物体的状态
int  attr;    // 物体的属性
int  mati; // 新增的，-1 表示没有材质
float *avg_radius;  // 包含 OBJECT4DV2_MAX_FRAMES 个元素
                    //物体的平均半径，用于碰撞检测
float *max_radius;  //包含 OBJECT4DV2_MAX_FRAMES 个元素，物体的最大半径

POINT4D world_pos;  // 物体在世界坐标系中的位置

VECTOR4D dir;    // 物体在局部坐标系下的旋转角度
         // 坐标或单位方向向量

VECTOR4D ux,uy,uz;  // 用于记录朝向的局部坐标轴
         // 物体被旋转时它们将自动更新

int num_vertices;   // 每帧包含的顶点数
int num_frames;     // 新增的，帧数
int total_vertices; // 全部顶点
int curr_frame;     // 当前帧，如果只有一帧，则为 0

VERTEX4DTV1_PTR vlist_local; // 局部顶点数组
VERTEX4DTV1_PTR vlist_trans; // 变换后的顶点数组

// 上述两个指针必须指向顶点列表的开头
VERTEX4DTV1_PTR head_vlist_local;
VERTEX4DTV1_PTR head_vlist_trans;

// 新增的，纹理坐标列表
POINT2D_PTR tlist;      // 3*最大多边形数

BITMAP_IMAGE_PTR texture; // 新增的，指向纹理信息的指针，用于简单纹理映射

int num_polys;          // 物体网格中的多边形数
POLY4DV2_PTR plist;     // 新增的，指向多边形的指针

// 方法 ////////////////////////////////////////////////////////

// 指定当前帧的功能非常重要，因此将其作为成员函数
// 调用其他函数之前如果没有指定当前帧可能会崩溃
int Set_Frame(int frame);

} OBJECT4DV2, *OBJECT4DV2_PTR;
```

根据其名称以及前面的多边形定义，读者应该能够明白大部分新增字段的含义。大部分被修改/新增的

字段都与顶点数组、纹理坐标数组以及多帧支持相关。下面介绍这些字段：

- **avg_radius、max_radius**——现在的物体包含多个帧，每帧中的顶点数据将各不相同，因此平均半径和最大半径随不同的帧而异。因此，avg_radius、max_radius 必须是数组。
- **num_frame**——动画帧数。
- **curr_frame**——当前活动的帧。
- **vlist_local**——指向活动帧的顶点网格的指针。
- **vlist_trans**——指向活动帧的顶点网格（变换后的版本）的指针。
- **head_vlist_local**——指向整个顶点列表的开头位置。
- **head_vlist_trans**——指向整个顶点列表（变换后的版本）的开头位置。
- **tlist**——指向纹理坐标数组开头的指针，对于所有的动画帧，只需要一个纹理坐标数组。另外，请花些时间来了解这个数组中包含的信息。这个数组用于存储整个网格的实际纹理坐标。从几何学上说，最多有 3*num_vertices 个纹理坐标；但实际上可能并没有这么多。例如，立方体网格可能只有纹理坐标（0，0）、（0，1）、（1，0）和（1，1），即总共 4 个纹理坐标；而每个多边形顶点只有一个指向纹理数组的索引，它指出了该顶点使用的纹理坐标。在最糟糕的情况下，每个顶点在包含它的每个多边形中，都有不同的纹理坐标，在这种情况下，将需要 3*num_vertices 个纹理坐标。

下面更详细地介绍网格动画以及上述字段的含义。正如前面指出的，我们初涉动画时，将使用静态多帧动画：加载一系列的网格，然后依次播放显示这些网格，就像实现 2D 位图动画那样。然而，我们对网格动画指定了两个限制条件：首先，所有网格包含的顶点数必须相同；其次，组成每个多边形的定义不能发生变化。惟一可以改变的是顶点的位置。通过使用这些规则，可以支持 Quake II .MD2 格式。本章不会实现任何动画功能，但将为以后实现这些功能做好准备。

提示：高级动画系统使用基于物理模型的前推或逆推运动学（forward or inverse kinematics）而不是静态网格信息。数学模型根据外力或动画数据对链杆进行变换，这样构成物体的各个子物体将分别运动，形成整个物体的动画。

选择当前帧后，将根据指针 head_vlist_*、num_vertices 和 curr_frame 来找到当前帧的地址。然后，将指针 vlist_local 和 vlist_trans 指向当前帧，这样对 vlist_local 和 vlist_trans 执行操作时操作的将是当前帧，如图 9.5 所示。除此之外，改变当前帧时，还必须存取平均半径和最大半径数组中相应的元素。vlist_local 和 vlist_trans 使得多帧支持非常简单——只是操纵指针而已。

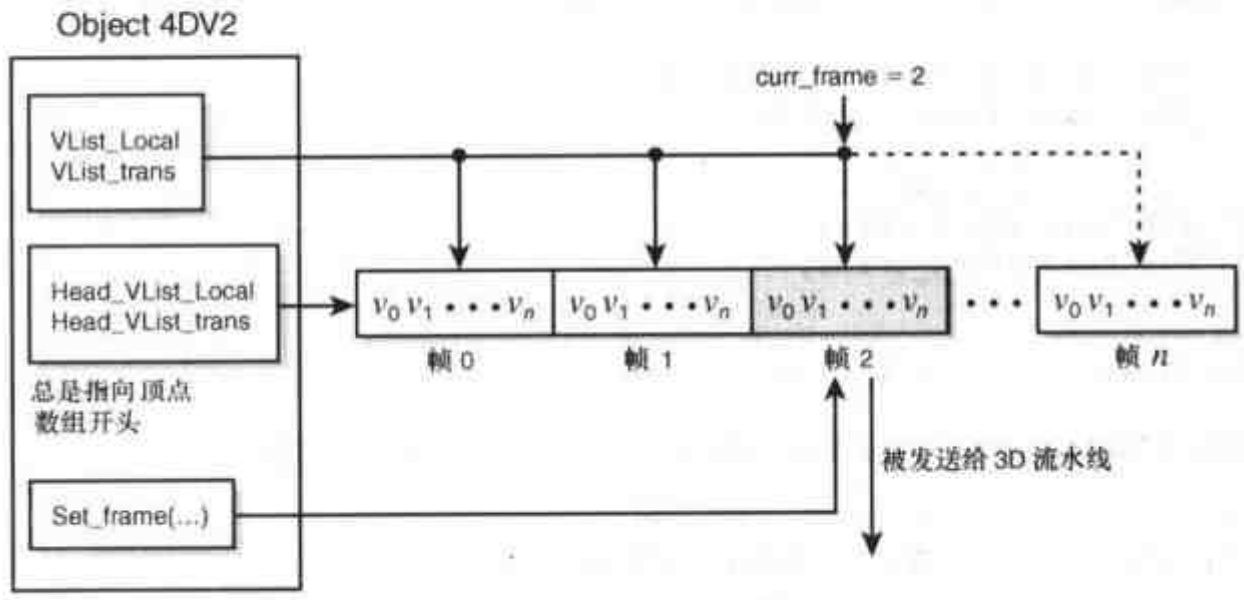


图 9.5　多帧支持的数据流程图

接下来，使用动态分配的数组来提高内存的使用效率以及帮助实现多帧支持。这种方法存在的惟一问题是，并非所有的文件格式都以类似于下面的内容打头：

```
"number of polygons, number of vertices, number of texture coordinates"
```

因此必须使用某种技巧来分配数组（并在读取文件中实际的值后重新分配数组）。例如，需要创建一个 OBJECT4DV2 对象，并加载顶点和多边形，但如果读取的是.COB 文件格式，一开始并不知道实际上有多少个纹

理坐标，因此估计在最糟糕的情况下有多少个顶点坐标。随便说一句，在分配内存空间时，如果不知道要分配多少，可按最糟糕的情况进行分配，以后再重新分配内存，复制原来分配的内存空间中的内容，并释放这些内存。

最后，还需要在 OBJECT4DV2 结构中新增一个方法，用于指定当前帧。如果在没有指定当前帧之前调用其他函数，可能导致系统崩溃。这个方法名为 Set_Frame()。要调用这个函数，只需这样做即可：

```
object.Set_Frame(0);
```

上述代码将当前帧设置为第 0 帧。为让读者了解操作指针有多简单，下面列出了该函数的代码：

```
// 指定当前帧的功能非常重要，因此将其作为成员函数
int OBJECT4DV2::Set_Frame(int frame)
{
// 这个函数指定多帧物体的当前帧
// 如果物体不是多帧的，这个函数将不起作用

// 检查物体是否有效
if (!this)
  return(0);

// 检查物体是否是多帧的
if (!(this->attr & OBJECT4DV2_ATTR_MULTI_FRAME))
  return(0);

// 物体有效且是多帧的，将指针指向当前帧数据

// 检查参数 frame 的值是否超界
if (frame < 0 )
  frame = 0;
else
if (frame >= this->num_frames)
  frame = this->num_frames - 1;

// 设置当前帧
this->curr_frame = frame;

// 让指针指向顶点列表中相应的帧

this->vlist_local = &(this->head_vlist_local[frame*this->num_vertices]);
this->vlist_trans = &(this->head_vlist_trans[frame*this->num_vertices]);

// 成功返回
return(1);

} // end Set_Frame
```

注意：还有一个外部函数版本：Set_OBJECT4DV2_Frame (OBJECT4DV2_PTR obj, int frame)，它与前面的版本相同，但是常规函数而不是对象的方法。

最后，每个物体都有一个纹理，该纹理被用于整个物体。多边形结构中的纹理指针指定一个用于多边形的纹理，而在物体中，指定一个用于整个物体的纹理。后面可能对此进行修改，但对物体进行纹理映射时，将纹理视为表层将更容易处理。加载物体时，将一个纹理图加载到 OBJECT4DV2 的纹理对象中，然后在加载多边形时，让多边形的内部纹理指针指向 OBJECT4DV2 的内部纹理。如果有材质库就好了，这样可以使用多个纹理，这将在以后再实现。

9.2.6　函数清单和原型

本章修改的函数很多，这里简要地介绍一下。这些函数大部分在以前介绍过，只是现在为 2.0 版。因此，这里只简要地介绍每个函数的用途，而不详细介绍其参数和调用范例，因为它们并非新增的，而只是版本不同而已，读者应该知道如何使用它们。这里介绍它们，旨在让读者知道新的引擎有哪些功能。

这些原型可能让读者感到迷惑，因为对于有些新函数，读者并不知道其功能，但作者仍希望读者仔细阅读其中的每个函数，这样后面讨论理论知识时，读者将对流水线的功能有大概的了解。另外，有些函数

只是用于测试，并非最终引擎的一部分——它们只适用于测试 OBJECT4DV1 和 RENDERLIST4DV1。本章不需要使用它们，作者在过渡到数据结构 2.0 版时编写了它们，将其用于测试。这里没有列出这些函数，但.CPP 和.H 文件中包含它们。下面是本章中一些很有用的函数原型，它们包含在引擎的 T3DLIB7.CPP|H 模块中，作者将一些用途相似的函数作为一组：

函数：

```
char *Extract_Filename_From_Path(char *filepath, char *filename);
```

用途：从完整的文件路径中提取文件名。

函数：

```
int Set_OBJECT4DV2_Frame(OBJECT4DV2_PTR obj, int frame);
```

用途：设置多网格动画中的当前帧。

函数：

```
int Destroy_OBJECT4DV2(OBJECT4DV2_PTR obj);
```

用途：删除物体，并释放其占用的内存。

函数：

```
int Init_OBJECT4DV2(OBJECT4DV2_PTR obj,  // object to allocate
           int _num_vertices,
           int _num_polys,
           int _num_frames,
           int destroy=0);
```

用途：初始化物体，并为其分配内存。

函数：

```
void Translate_OBJECT4DV2(OBJECT4DV2_PTR obj, VECTOR4D_PTR vt);
```

用途：在 3D 空间中平移物体。

函数：

```
void Scale_OBJECT4DV2(OBJECT4DV2_PTR obj, VECTOR4D_PTR vs, int all_frames=0);
```

用途：对物体的一个或所有帧进行缩放。

函数：

```
void Transform_OBJECT4DV2(OBJECT4DV2_PTR obj, MATRIX4X4_PTR mt,
           int coord_select,
           int transform_basis,
           int all_frames=0);
```

用途：使用传入的矩阵对物体的当前帧或所有帧进行变换。

函数：

```
void Rotate_XYZ_OBJECT4DV2(OBJECT4DV2_PTR obj,
           float theta_x,
           float theta_y,
           float theta_z,
           int all_frames);
```

用途：根据传入的参数对物体的局部坐标轴 x、y 和 z 进行旋转。

函数：

```
void Model_To_World_OBJECT4DV2(OBJECT4DV2_PTR obj,
      int coord_select = TRANSFORM_LOCAL_TO_TRANS, int all_frames=0);
```

用途：对传入的物体执行模型坐标到世界坐标变换。

函数：

```
int Cull_OBJECT4DV2(OBJECT4DV2_PTR obj, CAM4DV1_PTR cam, int cull_flags);
```

用途：在观察流水线中（准确地说是根据视景体）对物体进行剔除。

函数：

```
void Remove_Backfaces_OBJECT4DV2(OBJECT4DV2_PTR obj, CAM4DV1_PTR cam);
```

用途：根据传入的相机，对物体执行背面消除操作。

函数：

```
void Remove_Backfaces_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list, CAM4DV1_PTR cam);
```

用途：根据传入的相机，对渲染列表执行背面消除操作。

函数：

```
void Camera_To_Perspective_OBJECT4DV2(OBJECT4DV2_PTR obj, CAM4DV1_PTR cam);
```

用途：根据传入的相机，对物体执行相机坐标到透视坐标变换。

函数：

```
void Perspective_To_Screen_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                      CAM4DV1_PTR cam);
```

用途：根据传入的相机，对渲染列表执行透视坐标到屏幕坐标变换。

函数：

```
void Camera_To_Perspective_Screen_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                           CAM4DV1_PTR cam);
```

用途：根据传入的相机，对渲染列表执行相机坐标到屏幕坐标变换，相当于将透视变换和屏幕变换合而为一。

函数：

```
void Camera_To_Perspective_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                      CAM4DV1_PTR cam);
```

用途：根据传入的相机，对渲染列表执行相机坐标到透视坐标变换。

函数：

```
void World_To_Camera_OBJECT4DV2(OBJECT4DV2_PTR obj, CAM4DV1_PTR cam);
```

用途：根据传入的相机，对物体执行世界坐标到相机坐标变换。

函数：

```
void World_To_Camera_RENDERLIST4DV2(RENDERLIST4DV2_PTR render_list,
                    CAM4DV1_PTR cam);
```

用途：根据传入的相机，对渲染列表执行世界坐标到相机坐标变换。

函数：

```
void Camera_To_Perspective_Screen_OBJECT4DV2(OBJECT4DV2_PTR obj, CAM4DV1_PTR cam);
```

用途：根据传入的相机，对物体执行相机坐标到屏幕坐标变换，即将透视变换和屏幕变换合而为一。

函数：

```
void Perspective_To_Screen_OBJECT4DV2(OBJECT4DV2_PTR obj, CAM4DV1_PTR cam);
```

用途：根据传入的相机，对物体执行透视坐标到屏幕坐标变换。

函数：

```
int Insert_POLY4DV2_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                    POLY4DV2_PTR poly);

int Insert_POLYF4DV2_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                     POLYF4DV2_PTR poly);
```

用途：将传入的多边形插入到渲染列表中。

函数：

```
int Insert_OBJECT4DV2_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                     OBJECT4DV2_PTR obj,
```

```
                    int insert_local);
```

用途：将传入的物体分解为多边形，并将它们插入到渲染列表中。

函数：

```
void Reset_OBJECT4DV2(OBJECT4DV2_PTR obj);
```

用途：重置物体的状态值。

函数：

```
int Compute_OBJECT4DV2_Poly_Normals(OBJECT4DV2_PTR obj);
```

用途：计算传入的物体的多边形法线，供光照函数使用。

函数：

```
void Draw_OBJECT4DV2_Wire(OBJECT4DV2_PTR obj,
             UCHAR *video_buffer, int lpitch);

void Draw_OBJECT4DV2_Wire16(OBJECT4DV2_PTR obj,
              UCHAR *video_buffer, int lpitch);
```

用途：使用线框模式将传入的物体渲染到屏幕上。

函数：

```
void Draw_RENDERLIST4DV2_Wire(RENDERLIST4DV2_PTR rend_list,
              UCHAR *video_buffer, int lpitch);

void Draw_RENDERLIST4DV2_Wire16(RENDERLIST4DV2_PTR rend_list,
              UCHAR *video_buffer, int lpitch);
```

用途：使用线框模式将传入的渲染列表渲染到屏幕上。

函数：

```
void Draw_RENDERLIST4DV2_Solid(RENDERLIST4DV2_PTR rend_list,
              UCHAR *video_buffer, int lpitch);

void Draw_RENDERLIST4DV2_Solid16(RENDERLIST4DV2_PTR rend_list,
              UCHAR *video_buffer, int lpitch);
```

用途：将传入的渲染列表渲染到屏幕上，支持固定着色、恒定着色和纹理映射。

函数：

```
float Compute_OBJECT4DV2_Radius(OBJECT4DV2_PTR obj);
```

用途：计算物体中所有帧的平均半径和最大半径。

函数：

```
int Compute_OBJECT4DV2_Vertex_Normals(OBJECT4DV2_PTR obj);
```

用途：计算物体的顶点法线。

函数：

```
int Load_OBJECT4DV2_PLG(OBJECT4DV2_PTR obj, // 指向物体的指针
         char *filename,     // PLG 文件的名称
         VECTOR4D_PTR scale,   // 初始缩放因子
         VECTOR4D_PTR pos,    // 初始位置
         VECTOR4D_PTR rot,    // 初始旋转角度
         int vertex_flags=0); // 指定是否反转顶点顺序
```

用途：从磁盘中加载一个 PLG/PLX 格式的 3D 网格文件。

函数：

```
int Load_OBJECT4DV2_3DSASC(OBJECT4DV2_PTR obj,   // 指向物体的指针
             char *filename,    // ASC 文件的名称
             VECTOR4D_PTR scale,  // 初始缩放因子
             VECTOR4D_PTR pos,   // 初始位置
```

```
                VECTOR4D_PTR rot,   // 初始旋转角度
                int vertex_flags=0); // 指定是否反转顶点顺序
```

用途：从磁盘中加载一个 3D Studio Max .ASC 网格文件，参数 vertex_flags 指定是否要反转顶点环绕顺序。

函数：

```
int Load_OBJECT4DV2_COB(OBJECT4DV2_PTR obj,   // 指向物体的指针
          char *filename,    // Caligari COB 文件的名称
          VECTOR4D_PTR scale, // 初始缩放因子
          VECTOR4D_PTR pos,    // 初始位置
          VECTOR4D_PTR rot,  // 初始旋转角度
          int vertex_flags=0);  // 指定是否反转顶点的顺序
                  // 以及执行变换
```

用途：从磁盘中加载一个 Caligari trueSpace 3D .COB 网格文件，支持固定着色、恒定着色、Gouraud 着色和纹理映射。

函数：

```
void Reset_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list);
```

用途：重置传入的渲染列表，为下次流水线处理做准备。

函数：

```
int Light_OBJECT4DV2_World16(OBJECT4DV2_PTR obj,  // 要处理的物体
        CAM4DV1_PTR cam,  // 相机
        LIGHTV1_PTR lights,// 光源列表
        int max_lights);    // 列表包含的最大光源数

int Light_OBJECT4DV2_World(OBJECT4DV2_PTR obj,  // 要处理的物体
         CAM4DV1_PTR cam,    // 相机
         LIGHTV1_PTR lights, // 光源列表
         int max_lights);   // 列表包含的最大光源数
```

用途：根据传入的相机和光源列表，对物体执行光照处理；它们分别是 8 位版本和 16 位版本。

函数：

```
int Light_RENDERLIST4DV2_World16(RENDERLIST4DV2_PTR rend_list, // 要处理的
                                                              //渲染列表
       CAM4DV1_PTR cam,    // 相机
       LIGHTV1_PTR lights, // 光源列表
       int max_lights);   // 列表包含的最大光源数

int Light_RENDERLIST4DV2_World(RENDERLIST4DV2_PTR rend_list, // 要处理的
                                                             // 渲染列表
         CAM4DV1_PTR cam,     // 相机
         LIGHTV1_PTR lights,  // 光源列表
         int max_lights);     // 列表包含的最大光源数
```

用途：根据传入的相机和光源列表，对渲染列表执行光照处理；它们分别是 8 位版本和 16 位版本。

函数：

```
void Sort_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list, int sort_method);
```

用途：根据参数 sort_method 指定的排序方法，对渲染列表执行 z 排序。

函数：

```
int Compare_AvgZ_POLYF4DV2(const void *arg1, const void *arg2);

int Compare_NearZ_POLYF4DV2(const void *arg1, const void *arg2);

int Compare_FarZ_POLYF4DV2(const void *arg1, const void *arg2);
```

用途：排序算法使用的排序方法，Sort_RENDERLIST4DV2()将调用这些方法。

函数：

```
void Draw_Textured_Triangle(POLYF4DV2_PTR face,
```

```
            UCHAR *dest_buffer, int mem_pitch);

void Draw_Textured_Triangle16(POLYF4DV2_PTR face,
              UCHAR *dest_buffer, int mem_pitch);
```

用途：用于绘制三角形的低级仿射纹理渲染函数；它们分别是 8 位版本和 16 位版本。

函数：

```
void Draw_Textured_TriangleFS(POLYF4DV2_PTR face,   // 指向多边形面的指针
          UCHAR *_dest_buffer,// 指向视频缓存的指针
          int mem_pitch);    // 每行占据的字节数（320、640 等）

void Draw_Textured_TriangleFS16(POLYF4DV2_PTR face,   // 指向多边形面的指针
          UCHAR *_dest_buffer,   // 指向视频缓存的指针
          int mem_pitch);    // 每行占据的字节数（320、640 等）
```

用途：支持恒定着色的用于绘制三角形的低级仿射纹理渲染函数；它们分别是 8 位版本和 16 位版本。

函数：

```
void Draw_Gouraud_Triangle16(POLYF4DV2_PTR face,   // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch);   // 每行占据的字节数（320、640 等）

void Draw_Gouraud_Triangle(POLYF4DV2_PTR face,     // 指向多边形面的指针
          UCHAR *dest_buffer, // 指向视频缓存的指针
          int mem_pitch);   // 每行的字节数（320、640 等）
```

用途：支持 Gouraud 着色的用于绘制三角形的低级仿射纹理渲染函数；它们分别是 8 位版本和 16 位版本。

函数：

```
void Draw_Top_Tri2_16(float x1, float y1,
          float x2, float y2,
          float x3, float y3,
          int color,
          UCHAR *_dest_buffer, int mempitch);

void Draw_Bottom_Tri2_16(float x1, float y1,
             float x2, float y2,
             float x3, float y3,
             int color,
             UCHAR *_dest_buffer, int mempitch);

void Draw_Top_Tri2(float x1, float y1,
           float x2, float y2,
           float x3, float y3,
           int color,
           UCHAR *_dest_buffer, int mempitch);

void Draw_Bottom_Tri2(float x1, float y1,
             float x2, float y2,
             float x3, float y3,
             int color,
             UCHAR *_dest_buffer, int mempitch);
```

用途：绘制平底三角形和平顶三角形的低级函数，包括 8 位和 16 位版本。

函数：

```
void Draw_Triangle_2D2_16(float x1, float y1,
             float x2, float y2,
             float x3, float y3,
             int color,
             UCHAR *dest_buffer, int mempitch);

void Draw_Triangle_2D2(float x1, float y1,
             float x2, float y2,
             float x3, float y3,
             int color,
```

```
                UCHAR *dest_buffer, int mempitch);
```

用途：绘制常规三角形的低级函数，包括 8 位和 16 位版本。

函数：

```
int Load_Bitmap_File2(BITMAP_FILE_PTR bitmap, char *filename);
```

用途：位图加载函数，支持文件格式.BMP 和.PCX，将其加载到位图文件对象中。

函数：

```
int Load_Bitmap_PCX_File(BITMAP_FILE_PTR bitmap, char *filename);
```

用途：将.PCX 文件加载到位图文件对象中。

9.3　重新编写物体加载函数

作者将不断地重新编写加载函数，这是一项非常麻烦的工作。令人欣慰的是，充分利用文件格式本身后，可使加载函数达到比较完善的程度。.PLG/PLX 和.ASC 格式就属于这种情况，即它们根本不支持引擎新增的特性，因此对于这些格式的加载函数，需要做的修改不多。这里之所以保留它们，是因为它们易于手工处理。另一方面，Caligari .COB 格式能够满足我们的所有需求。

.PLG/PLX 格式支持着色模式（但不支持纹理映射），另外它非常简单，主要用于手工建立物体模型，以便对引擎进行测试。最后，.PLG 格式对 8/16 位的硬编码支持非常麻烦，几乎让作者发疯，但编写 3D 引擎时，从这种格式开始着手是不错的选择，因此这里保留了它。

.ASC 格式只能提供网格数据和颜色，它不支持纹理映射和光照模型等。为支持这些特性（至少是光照），在加载函数中使用了参数 vertex_flags 来指定着色方法。然而，.ASC 和 PLG/PLX 并不支持纹理映射，因为我们无法指定纹理坐标。.COB 格式支持我们所需的各种特性，只有让建模程序使用这种格式导出 3D 模型，才能提供可供引擎使用的光照信息和纹理信息。直觉告诉我，随着本书介绍的深入，将越来越多地使用.COB 格式，越来越少地使用.PLG/PLX 和.ASC 格式。下面来看看新的加载函数及其支持的特性。

9.3.1　更新.PLG/PLX 加载函数

对.PLG/PLX 加载函数所做的修改最少，但这里只列出它的代码，因为它不算太长。所做的修改主要包括对 OBJECT4DV2 结构的支持、访问数据元素的方式以及设置一些新的顶点标记。新结构 VERTEX4DV1 包含一些属性标记，用于指定顶点是否包含 3D 点、法线和纹理坐标。这里介绍对.PLG/PLX 加载函数所做的修改，以便读者在阅读附带光盘中其他的加载函数时不会迷失方向。下面是这个函数的代码，其中重要的代码用粗体显示：

```
int Load_OBJECT4DV2_PLG(OBJECT4DV2_PTR obj, // 指向物体的指针
           char *filename,   // PLG 文件的名称
           VECTOR4D_PTR scale, // 初始缩放因子
           VECTOR4D_PTR pos,   // 初始位置
           VECTOR4D_PTR rot, // 初始旋转角度
           int vertex_flags) // 顶点标记，用于覆盖某些设置
{
// 这个函数从磁盘中加载 PLG 物体
// 并允许指定非动态物体的位置，对其进行缩放和旋转，以免以后再调用这些函数
// 由于只有一帧，因此加载物体并相应地设置 OBJECT4DV2 的字段

FILE *fp;      // 文件指针
char buffer[256];  // 工作缓冲区

char *token_string;  // 指向从文件中读取的文本的指针，以便对其进行分析

// 回顾 PLG 文件格式
// # 这是注释

// # 物体描述符
// object_name_string num_verts_int num_polys_int
```

```
// # 顶点列表
// x0_float y0_float z0_float
// x1_float y1_float z1_float
// x2_float y2_float z2_float
// .
// .
// xn_float yn_float zn_float
//
// # 多边形列表
// surface_description_ushort num_verts_int v0_index_int v1_index_int .. // .
// .
// surface_description_ushort num_verts_int v0_index_int v1_index_int ..

// 假设每个元素占据一行
// 因此首先找到并读取物体描述符
// 然后找到并读取顶点列表，最后找到并读取多边形列表

// 第 1 步：清空物体的内容并对其进行初始化
memset(obj, 0, sizeof(OBJECT4DV2));

// 将物体的状态设置为活动和可见
obj->state = OBJECT4DV2_STATE_ACTIVE | OBJECT4DV2_STATE_VISIBLE;

// 设置物体的位置
obj->world_pos.x = pos->x;
obj->world_pos.y = pos->y;
obj->world_pos.z = pos->z;
obj->world_pos.w = pos->w;

// 设置物体包含的帧数
obj->num_frames = 1;
obj->curr_frame = 0;
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME;

// 第 2 步：打开文件以便进行读取
if (!(fp = fopen(filename, "r")))
{
Write_Error("Couldn't open PLG file %s.", filename);
return(0);
} // end if

// 第 3 步：读取第一行字符串，这是物体描述符
if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
{
Write_Error("PLG file error with file %s (object descriptor invalid).",
      filename);
return(0);
} // end if

Write_Error("Object Descriptor: %s", token_string);

// 分析物体描述符
sscanf(token_string, "%s %d %d",
    obj->name, &obj->num_vertices, &obj->num_polys);

// 为存储顶点数和多边形数的变量分配内存
if (!Init_OBJECT4DV2(obj,  // 为物体分配内存
        obj->num_vertices,
        obj->num_polys,
        obj->num_frames))
 {
 Write_Error("\nPLG file error with file %s (can't allocate memory).",
       filename);
 } // end if

// 第 4 步：加载顶点列表
```

```
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
  // 读取下一个顶点
  if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
    {
    Write_Error("PLG file error with file %s (vertex list invalid).",
          filename);
    return(0);
    } // end if

  // 分析顶点数据
  sscanf(token_string, "%f %f %f", &obj->vlist_local[vertex].x,
                   &obj->vlist_local[vertex].y,
                   &obj->vlist_local[vertex].z);
  obj->vlist_local[vertex].w = 1;

  // 执行缩放操作
  obj->vlist_local[vertex].x*=scale->x;
  obj->vlist_local[vertex].y*=scale->y;
  obj->vlist_local[vertex].z*=scale->z;

  Write_Error("\nVertex %d = %f, %f, %f, %f", vertex,
              obj->vlist_local[vertex].x,
              obj->vlist_local[vertex].y,
              obj->vlist_local[vertex].z,
              obj->vlist_local[vertex].w);

  // 每个顶点至少包含一个 3D 点，因此相应地设置属性
  SET_BIT(obj->vlist_local[vertex].attr, VERTEX4DTV1_ATTR_POINT);

  } // end for vertex

// 计算平均半径和最大半径
Compute_OBJECT4DV2_Radius(obj);

Write_Error("\nObject average radius = %f, max radius = %f",
      obj->avg_radius, obj->max_radius);

int poly_surface_desc = 0; // PLG/PLX 面描述符
int poly_num_verts    = 0; // 当前多边形的顶点数(总是为 3)
char tmp_string[8];     // 用于存储面描述符的字符串，将检测其值是否为十六进制的

// 第 5 步：加载顶点列表
for (int poly=0; poly < obj->num_polys; poly++)
  {
  // 读取下一个多边形描述符
  if (!(token_string = Get_Line_PLG(buffer, 255, fp)))
  {
  Write_Error("PLG file error with file %s
         (polygon descriptor invalid).",filename);
  return(0);
  } // end if

  Write_Error("\nPolygon %d:", poly);

  // 每个多边形都有 3 个顶点
  // 因为我们规定所有模型都由三角形组成
  // 读取面描述符、顶点数和顶点列表
  sscanf(token_string, "%s %d %d %d %d", tmp_string,
          &poly_num_verts, // 总为 3
          &obj->plist[poly].vert[0],
          &obj->plist[poly].vert[1],
          &obj->plist[poly].vert[2]);

// 我们允许面描述符为十六进制，在这种情况下，面描述符以"0x"打头
// 对此进行检测
if (tmp_string[0] == '0' && toupper(tmp_string[1]) == 'X')
```

```
  sscanf(tmp_string,"%x", &poly_surface_desc);
 else
  poly_surface_desc = atoi(tmp_string);

  // 将多边形的顶点列表指针指向物体的顶点列表
  obj->plist[poly].vlist = obj->vlist_local;

  Write_Error("\nSurface Desc = 0x%.4x, num_verts = %d,
        vert_indices [%d, %d, %d]",
                poly_surface_desc,
                poly_num_verts,
                obj->plist[poly].vert[0],
               obj->plist[poly].vert[1],
               obj->plist[poly].vert[2]);

  // 设置顶点列表和多边形的顶点索引后
  // 对面描述符进行分析，以相应地设置多边形的字段

  // 提取面描述符中的数据
  // 首先提取单面/双面设置
  if ((poly_surface_desc & PLX_2SIDED_FLAG))
    {
    SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_2SIDED);
    Write_Error("\n2 sided.");
    } // end if
  else
    {
    // 单面
    Write_Error("\n1 sided.");
    } // end else

  // 提取颜色模式和颜色值
  if ((poly_surface_desc & PLX_COLOR_MODE_RGB_FLAG))
    {
    // RGB 4.4.4 模式
    SET_BIT(obj->plist[poly].attr,POLY4DV2_ATTR_RGB16);

    // 提取颜色值，并将其复制到多边形的颜色字段中
    int red   = ((poly_surface_desc & 0x0f00) >> 8);
    int green = ((poly_surface_desc & 0x00f0) >> 4);
    int blue  = (poly_surface_desc & 0x000f);

    // 文件中的颜色数据总是 4.4.4 格式的，但图形卡为 5.5.5 或 5.6.5 格式
    // 虚拟颜色系统将 8.8.8 格式转换为 5.5.5 或 5.6.5 格式
    // 因此我们需要将 4.4.4 值转换为 8.8.8 格式
    obj->plist[poly].color = RGB16Bit(red*16, green*16, blue*16);
    Write_Error("\nRGB color = [%d, %d, %d]", red, green, blue);
    } // end if
  else
    {
    // 多边形使用的是颜色索引模式
    SET_BIT(obj->plist[poly].attr,POLY4DV2_ATTR_8BITCOLOR);

    // 提取最后 8 位，它们是颜色索引
    obj->plist[poly].color = (poly_surface_desc & 0x00ff);

    Write_Error("\n8-bit color index = %d", obj->plist[poly].color);

    } // end else

  // 处理着色模式
  int shade_mode = (poly_surface_desc & PLX_SHADE_MODE_MASK);

  // 设置多边形的着色模式
  switch(shade_mode)
     {
     case PLX_SHADE_MODE_PURE_FLAG: {
```

```
SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_SHADE_MODE_PURE);
Write_Error("\nShade mode = pure");
} break;

case PLX_SHADE_MODE_FLAT_FLAG: {
SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_SHADE_MODE_FLAT);
Write_Error("\nShade mode = flat");

} break;

case PLX_SHADE_MODE_GOURAUD_FLAG: {
SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_SHADE_MODE_GOURAUD);

// 多边形的顶点必须包含法线，因此相应地设置顶点的属性
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[0] ].attr,
    VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[1] ].attr,
    VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[2] ].attr,
    VERTEX4DTV1_ATTR_NORMAL);

Write_Error("\nShade mode = gouraud");
} break;

case PLX_SHADE_MODE_PHONG_FLAG: {
SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_SHADE_MODE_PHONG);

// 多边形的顶点必须包含法线，因此相应地设置顶点的属性
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[0] ].attr,
    VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[1] ].attr,
    VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[2] ].attr,
    VERTEX4DTV1_ATTR_NORMAL);

Write_Error("\nShade mode = phong");
} break;

default: break;
} // end switch

// 设置材质模式，以模拟 1.0 版
SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_DISABLE_MATERIAL);

// 将多边形的状态设置为活动的
obj->plist[poly].state = POLY4DV2_STATE_ACTIVE;


// 将多边形的顶点列表指针指向物体的顶点列表
obj->plist[poly].vlist = obj->vlist_local;

// 设置纹理坐标列表
obj->plist[poly].tlist = obj->tlist;


} // end for poly

// 计算多边形法线的长度
Compute_OBJECT4DV2_Poly_Normals(obj);

// 对于使用 Gouraud 着色的多边形，计算其顶点法线
Compute_OBJECT4DV2_Vertex_Normals(obj);

// 关闭文件
fclose(fp);

// 成功返回
```

```
return(1);

} // end Load_OBJECT4DV2_PLG
```

下面讨论各段用粗体显示的代码，它们是新增或经过修改的。从开头开始，第一段粗体代码如下：

```
// 设置帧数
obj->num_frames = 1;
obj->curr_frame = 0;
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME;
```

这段代码提供多帧支持，将当前帧设置为 0；对于单帧物体也必须执行这些操作。接下来为 OBJECT4DV2 分配内存：

```
if (!Init_OBJECT4DV2(obj,  // object to allocate
        obj->num_vertices,
        obj->num_polys,
        obj->num_frames))
 {
 Write_Error("\nPLG file error with file %s (can't allocate memory).",
        filename);
 } // end if
```

函数 Init_OBJECT4DV2()释放 OBJECT4DV2 原来占用的内存，并为其重新分配内存，代码如下：

```
int Init_OBJECT4DV2(OBJECT4DV2_PTR obj,  // 要为其分配内存的物体
           int _num_vertices,
           int _num_polys,
           int _num_frames,
           int destroy)
{
// 这个函数根据传入的参数值为 OBJECT4DV2 分配内存

// 首先删除物体（如果已经存在的话）
if (destroy)
  Destroy_OBJECT4DV2(obj);

// 首先为顶点列表分配内存
if (!(obj->vlist_local = (VERTEX4DTV1_PTR)malloc(sizeof(VERTEX4DTV1)*
                    num_vertices*_num_frames)))
  return(0);

// 清除数据
memset((void *)obj->vlist_local,0,sizeof(VERTEX4DTV1)*
                num_vertices*_num_frames);

if (!(obj->vlist_trans = (VERTEX4DTV1_PTR)malloc(sizeof(VERTEX4DTV1)*
                    num_vertices*_num_frames)))
  return(0);

// 清除数据
memset((void *)obj->vlist_trans,0,sizeof(VERTEX4DTV1)*
                  num_vertices*_num_frames);

// 纹理坐标数总是 3*多边形数
if (!(obj->tlist = (POINT2D_PTR)malloc(sizeof(POINT2D)*_num_polys*3)))
  return(0);

// 清除数据
memset((void *)obj->tlist,0,sizeof(POINT2D)*_num_polys*3);


// 为半径数组分配内存
if (!(obj->avg_radius = (float *)malloc(sizeof(float)*_num_frames)))
  return(0);

// 清除数据
memset((void *)obj->avg_radius,0,sizeof(float)*_num_frames);
```

```
if (!(obj->max_radius = (float *)malloc(sizeof(float)*_num_frames)))
  return(0);

// 清除数据
memset((void *)obj->max_radius,0,sizeof(float)*_num_frames);

// 为多边形列表分配内存
if (!(obj->plist = (POLY4DV2_PTR)malloc(sizeof(POLY4DV2)*_num_polys)))
  return(0);

// 清除数据
memset((void *)obj->plist,0,sizeof(POLY4DV2)*_num_polys);

// 设置指针
obj->head_vlist_local = obj->vlist_local;
obj->head_vlist_trans = obj->vlist_trans;

// 设置一些内部变量
obj->num_frames    = _num_frames;
obj->num_polys     = _num_polys;
obj->num_vertices = _num_vertices;
obj->total_vertices = _num_vertices*_num_frames;

// 成功返回
return(1);

} // end Init_OBJECT4DV2
```

读者可能注意到了，该函数调用了 Destroy_OBJECT4DV2()，后者的代码如下：

```
int Destroy_OBJECT4DV2(OBJECT4DV2_PTR obj)   // 要删除的物体
{
// 这个函数删除传入的物体，这基本上是释放其占用的内存

// 局部顶点列表
if (obj->head_vlist_local)
   free(obj->head_vlist_local);

// 变换后的顶点列表
if (obj->head_vlist_trans)
  free(obj->head_vlist_trans);

// 纹理坐标列表
if (obj->tlist)
  free(obj->tlist);

// 多边形列表
if (obj->plist)
  free(obj->plist);

// 物体半径数组
if (obj->avg_radius)
  free(obj->avg_radius);

if (obj->max_radius)
  free(obj->max_radius);

// 清除物体的数据
memset((void *)obj, 0, sizeof(OBJECT4DV2));

// 成功返回
return(1);

} // end Destroy_OBJECT4DV2
```

和以往一样，这里有大量的错误检查代码，这可能有些小题大做，但并没有坏处。回到.PLG 加载函数，

接下来的一段重要代码设置顶点的属性，指出其包含一个 3D 点：

```
// 顶点至少包含一个 3D 点，相应地设置其属性
SET_BIT(obj->vlist_local[vertex].attr, VERTEX4DTV1_ATTR_POINT);
```

对于每个包含 3D 点的顶点（几乎是全部顶点）都必须这样做，新的灵活的顶点格式允许顶点包含 3D 点、法线和纹理坐标。接下来的一段重要代码用于支持 Gouraud 和 Phong 着色，它们设置多边形的每个顶点，使之能够包含法线，如下所示：

```
// 多边形的顶点必须包含法线，相应地设置顶点属性
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[0] ].attr,
                    VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[1] ].attr,
                    VERTEX4DTV1_ATTR_NORMAL);
SET_BIT(obj->vlist_local[ obj->plist[poly].vert[2] ].attr,
                    VERTEX4DTV1_ATTR_NORMAL);
```

对于 Gouraud 着色和 Phong 着色（可能不会实现这种着色），必须知道每个顶点的法线，因此这里设置相应的属性，使顶点能够包含法线。最后，调用了计算多边形法线和顶点法线的函数。计算多边形法线的函数在前面介绍过：

```
// 计算多边形法线的长度
Compute_OBJECT4DV2_Poly_Normals(obj);
```

下面的函数计算每个多边形的顶点法线：

```
// 对于使用 Gouraud 着色的多边形，计算其顶点法线
Compute_OBJECT4DV2_Vertex_Normals(obj);
```

下面详细讨论这个新的.PLG/PLX 文件加载函数。

计算用于光照处理的顶点法线

有关光照计算和 Gouraud 着色算法将在下一节介绍，现在暂时不去管它，将注意力集中在顶点法线的计算机制上。问题如下：给定由 p 个多边形和 v 个顶点组成的模型，计算每个顶点的法线。但是，顶点法线的定义是什么呢？对光照处理而言，需要将多边形法线进行平均来计算顶点法线。图 9.6 显示了有一个简单网格以及每个多边形的多边形法线。要计算顶点法线，需要找出包含给定顶点的所有多边形，并根据每个多边形的面积对多边形法线（这里假定为单位法线——译者注）进行加权平均。

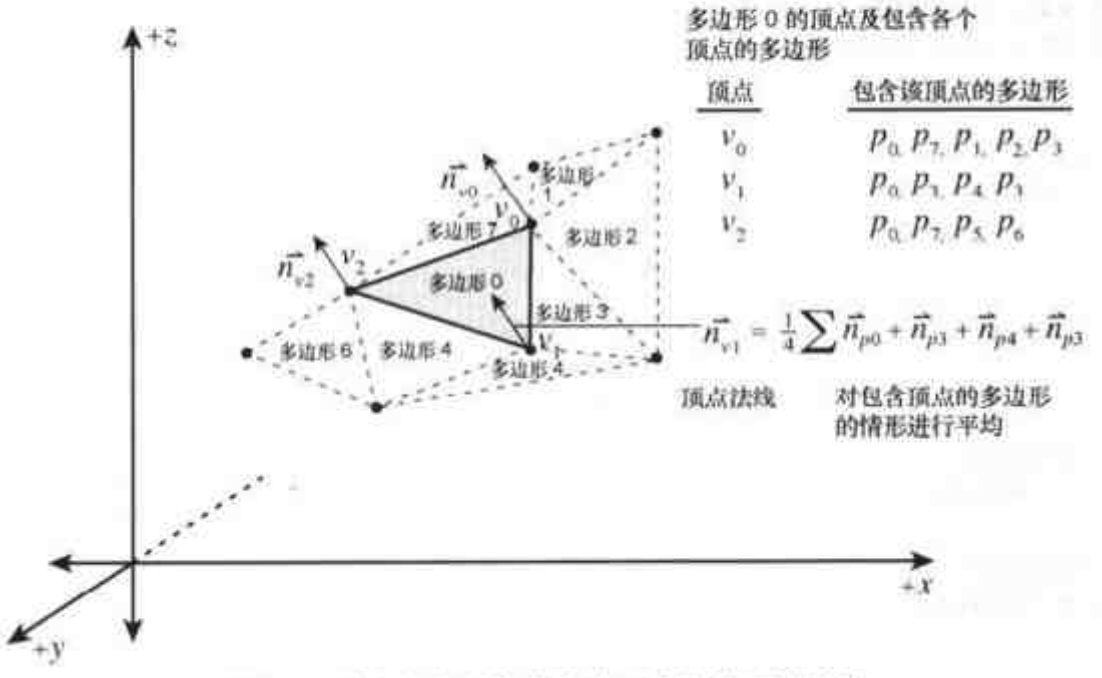


图 9.6　确定包含顶点的多边形并计算顶点法线

例如，假设顶点由两个多边形共享，如图 9.7 所示。多边形 1 的法线权重应比多边形 2 的法线权重大，因为前者的面积比后者大。事实上，在编码中这很容易实现。使用三角形任何两个向量的叉积来计算其法线时，法线的长度与三角形的面积成正比。事实上，法线的长度与这两个向量对应的平行四边形的面积相等，如图 9.8 所示。

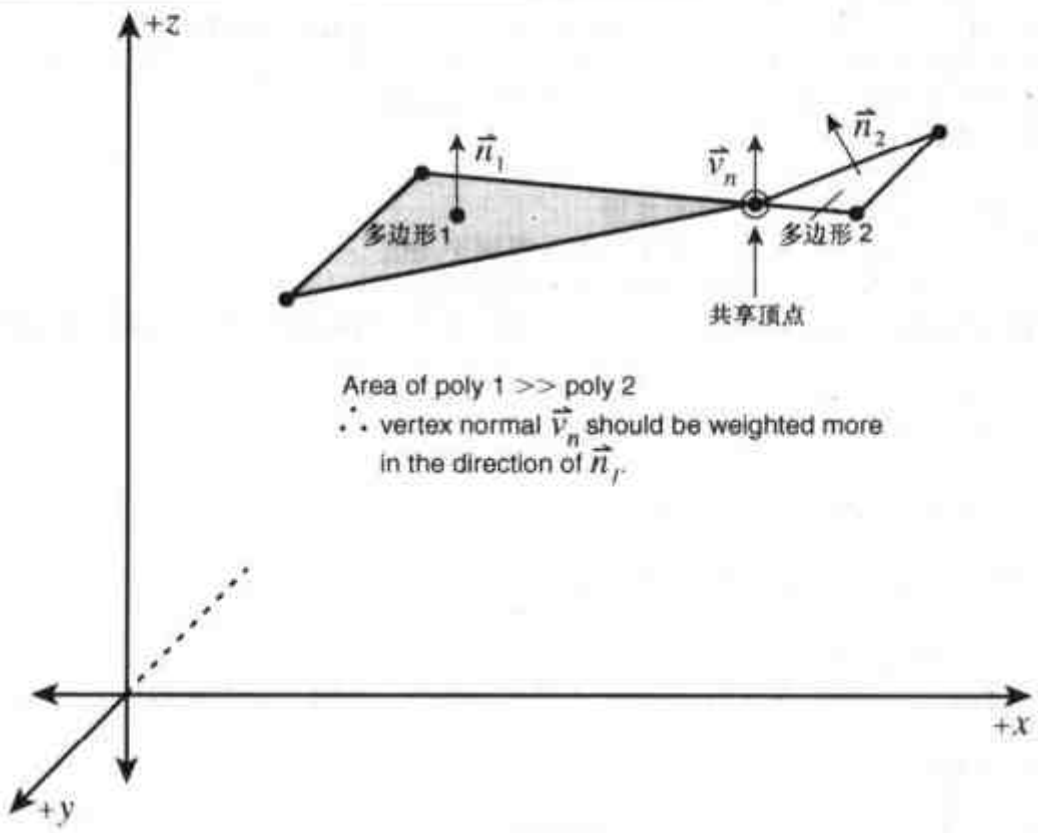


图 9.7　根据面积计算多边形法线的权重

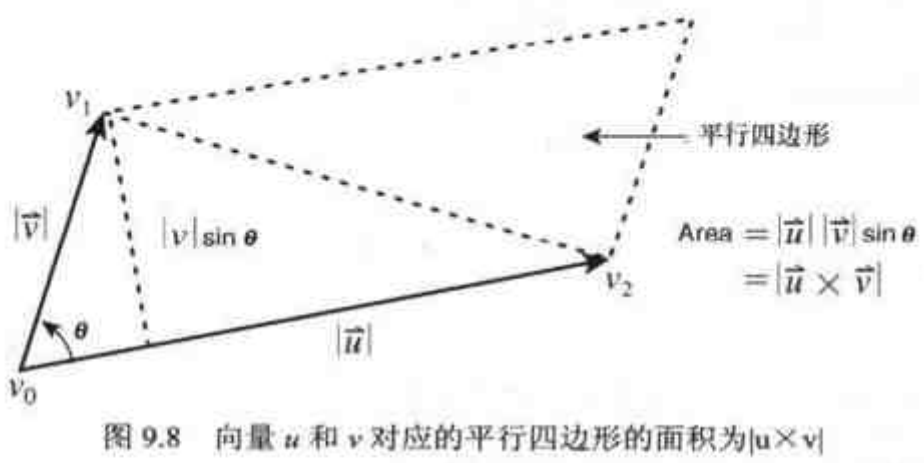


图 9.8　向量 u 和 v 对应的平行四边形的面积为$|u \times v|$

因此，向量 u 和 v 对应的三角形的面积为$|u \times v|/2$。因此，计算顶点法线的算法如下：

```
对于网格中的每个顶点 v_i
    v_sum_normal = <0, 0, 0>

    对于包含该顶点的每个多边形 p_i
        v_sum_normal += p_i 的法线
        下一个多边形
    归一化 v_sum_normal
下一个顶点
```

计算并得到顶点法线后，每当对模型进行变换时，必须对顶点法线也进行变换，以确保它们与模型同步；否则顶点法线将毫无用处。这增加了顶点变换操作的计算量，也就是说必须对顶点法线向量也进行变换；这相当于顶点数增加了一倍。不过，仅对采用 Gouraud 着色的多边形来说，才会出现这种情况；如果模型中采用 Gouraud 着色的多边形很少或者没有，这种开销将很小。

下面是在加载物体时离线计算顶点法线的函数：

```
int Compute_OBJECT4DV2_Vertex_Normals(OBJECT4DV2_PTR obj)
{
// 很多函数都需要使用多边形的顶点法线，
// 其中最重要的是对使用 Gouraud 着色的多变形执行光照计算
// 我们只为使用 Gouraud 着色的多边形计算顶点法线
// 对于每个顶点，确定共用该顶点的多边形，然后对这些多边形的法线进行平均，得到顶点法线
// 为判断是否需要考虑多边形法线，需检查多边形的着色模式

// 物体是否有效
if (!obj)
  return(0);
```

```
// 算法如下：遍历多边形列表中的每个多边形，判断它使用的是否是 Gouraud 着色，
// 如果是，则将其面法线累积到其顶点法线中，并将记录有多少个多边形包含该顶点的计数器加 1
// 遍历完毕后，根据计数器的值对累积值进行平均，得到顶点法线
// 这种算法的复杂度为 O(c*n)，而不是 O(n^2)

// 下述数组用于记录各个顶点有多少个多边形共用
// 遍历结束后，将累积的顶点法线值除以计数器的值，得到平均值
int polys_touch_vertex[OBJECT4DV2_MAX_VERTICES];
memset((void *)polys_touch_vertex, 0, sizeof(int)*OBJECT4DV2_MAX_VERTICES);

// 遍历物体的多边形列表，计算多边形的法线
//将其累积到顶点法线中，并将共用顶点的多边形数加 1

for (int poly=0; poly < obj->num_polys; poly++)
  {
  Write_Error("\nprocessing poly %d", poly);

  // 检查多边形是否需要顶点法线
  if (obj->plist[poly].attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
    {
    // 提取指向顶点列表的顶点索引
    // 多边形不是自包含的，而是基于物体的顶点列表的
    int vindex_0 = obj->plist[poly].vert[0];
    int vindex_1 = obj->plist[poly].vert[1];
    int vindex_2 = obj->plist[poly].vert[2];

    Write_Error("\nTouches vertices: %d, %d, %d",
          vindex_0, vindex_1, vindex_2);

    // 需要考虑该多边形的法线
    // 顶点是按顺时针方向排列的，因此 u=p0->p1，v=p0->p2，n=uxv
    VECTOR4D u, v, n;

    // 计算 u 和 v
    VECTOR4D_Build(&obj->vlist_local[ vindex_0 ].v,
          &obj->vlist_local[ vindex_1 ].v, &u);
    VECTOR4D_Build(&obj->vlist_local[ vindex_0 ].v,
          &obj->vlist_local[ vindex_2 ].v, &v);

    // 计算叉积
    VECTOR4D_Cross(&u, &v, &n);

    // 将共用顶点的多边形数加 1
    polys_touch_vertex[vindex_0]++;
    polys_touch_vertex[vindex_1]++;
    polys_touch_vertex[vindex_2]++;

  Write_Error("\nPoly touch array v[%d] = %d,  v[%d] = %d,  v[%d] = %d",
        vindex_0, polys_touch_vertex[vindex_0],
        vindex_1, polys_touch_vertex[vindex_1],
        vindex_2, polys_touch_vertex[vindex_2]);

    // 将多边形法线累积到顶点法线中
    // 这里没有对多边形法线进行归一化
    // 因为多边形法线的长度为向量 u 和 v 对应的平行四边形的面积
    // 它将被用作加权平均的权重
    VECTOR4D_Add(&obj->vlist_local[vindex_0].n, &n,
          &obj->vlist_local[vindex_0].n);
    VECTOR4D_Add(&obj->vlist_local[vindex_1].n, &n,
          &obj->vlist_local[vindex_1].n);
    VECTOR4D_Add(&obj->vlist_local[vindex_2].n, &n,
          &obj->vlist_local[vindex_2].n);
    } // end for poly

  } // end if needs vertex normals

// 计算出所有顶点法线的累积值后，需要对其进行平均
```

```
for (int vertex = 0; vertex < obj->num_vertices; vertex++)
  {
Write_Error("\nProcessing vertex: %d, attr: %d, contributors: %d", vertex,
                 obj->vlist_local[vertex].attr,
                 polys_touch_vertex[vertex]);

  // 检查顶点是否包含法线，如果有，则对其进行平均
  if (polys_touch_vertex[vertex] >= 1)
    {
    obj->vlist_local[vertex].nx/=polys_touch_vertex[vertex];
    obj->vlist_local[vertex].ny/=polys_touch_vertex[vertex];
    obj->vlist_local[vertex].nz/=polys_touch_vertex[vertex];

    // 将法线归一化
    VECTOR4D_Normalize(&obj->vlist_local[vertex].n);

    Write_Error("\nAvg Vertex normal: [%f, %f, %f]",
          obj->vlist_local[vertex].nx,
          obj->vlist_local[vertex].ny,
          obj->vlist_local[vertex].nz);

    } // end if

  } // end for

// 成功返回
return(1);

} // end Compute_OBJECT4DV2_Vertex_Normals
```

运行演示程序时，分析结果将被保存到文件 ERROR.TXT 中。读者可以通过查看该文件来了解顶点分析阶段发生的情况。

9.3.2　更新 3D Studio .ASC 加载函数

与前一版相比，.ASC 加载函数的变化不大，因为它不支持纹理、光照处理等。惟一新增的内容是对 OBJECT4DV2 的支持，这在前面讨论.PLG/PLX 加载函数时介绍过。然而，为提高该加载函数的功能，作者增加了参数 vertex_flags，用于指定物体的光照模型。这个函数的代码太长，这里无法列出，但和.PLG/PLX 加载函数一样，这个函数也新增了初始化物体和设置顶点类型的功能。该函数的原型如下：

```
int Load_OBJECT4DV2_3DSASC(OBJECT4DV2_PTR obj, // 指向物体的指针
           char *filename,    // ASC 文件的名称
           VECTOR4D_PTR scale, // 初始缩放因子
           VECTOR4D_PTR pos,  // 初始位置
           VECTOR4D_PTR rot,  // 初始旋转角度
           int vertex_flags)  // 指定是否需要反转顶点顺序以及设置着色方法
```

新增的功能都是基于参数 vertex_flags 的，下面是为设置该参数而定义的常量：

```
// 新增的用于简单模型格式的常量，用于设置着色方法
#define VERTEX_FLAGS_OVERRIDE_MASK   0xf000 // 用于提取参数值的掩码
#define VERTEX_FLAGS_OVERRIDE_CONSTANT 0x1000
#define VERTEX_FLAGS_OVERRIDE_EMISSIVE 0x1000 //别名
#define VERTEX_FLAGS_OVERRIDE_PURE   0x1000
#define VERTEX_FLAGS_OVERRIDE_FLAT   0x2000
#define VERTEX_FLAGS_OVERRIDE_GOURAUD 0x4000
#define VERTEX_FLAGS_OVERRIDE_TEXTURE 0x8000
```

正如读者看到的，这里定义了用于指定各种着色方法的常量。当前，只支持固定着色、恒定着色和 Gouraud 着色，如该函数中的下述代码所示：

```
// 首先检查是否指定了着色方法
int vertex_overrides = (vertex_flags & VERTEX_FLAGS_OVERRIDE_MASK);

if (vertex_overrides)
```

```
{
// 哪种着色方法
if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_PURE)
  SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_SHADE_MODE_PURE);

if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_FLAT)
  SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_SHADE_MODE_FLAT);

if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_GOURAUD)
  {
  SET_BIT(obj->plist[poly].attr, POLY4DV2_ATTR_SHADE_MODE_GOURAUD);

  // 需要顶点法线
  SET_BIT(obj->vlist_local[ obj->plist[poly].vert[0] ].attr,
      VERTEX4DTV1_ATTR_NORMAL);
  SET_BIT(obj->vlist_local[ obj->plist[poly].vert[1] ].attr,
      VERTEX4DTV1_ATTR_NORMAL);
  SET_BIT(obj->vlist_local[ obj->plist[poly].vert[2] ].attr,
      VERTEX4DTV1_ATTR_NORMAL);
  } // end if

  if (vertex_overrides & VERTEX_FLAGS_OVERRIDE_TEXTURE)
    SET_BIT(obj->plist[poly].attr,POLY4DV2_ATTR_SHADE_MODE_TEXTURE);

  } // end if
```

注意，多边形采用 Gouraud 着色时，将设置其顶点的属性标记 VERTEX4DV1_ATTR_NORMAL，因为在这种情况下需要使用顶点法线。下面的代码演示了如何在加载物体时，将着色方法设置为 Gouraud：

```
Load_OBJECT4DV2_3DSASC(&obj_player, "tie01.asc",
            &vscale, &vpos, &vrot,
            VERTEX_FLAGS_OVERRIDE_GOURAUD);
```

9.3.3 更新 Caligari .COB 加载函数

最后要介绍的是新的.COB 物体加载函数。这个函数的代码非常长，但大部分修改与支持新的 OBJECT4DV2 结构有关。然而，.COB 文件支持纹理坐标和纹理，因此必须在加载函数中添加这种支持。

介绍其代码之前，先来回顾一下我们是处理.COB 文件的。我们根据“Shader class: color”后面一行的内容来确定是否使用纹理映射；然后根据“Shader class: reflectance”后面一行的内容来确定采用固定着色、恒定着色还是 Gouraud 着色。有趣的是，使用纹理时，仍然需要指定着色模式。这个版本的引擎支持对纹理映射多边形进行固定着色和恒定着色。使用 Gouraud 着色时，需要进行插值计算，这将极大地降低纹理映射的速度。这意味着创建.COB 格式的模型时，对于要进行纹理映射的多边形，必须将其颜色设置为“color: texture map”，同时将其反射（reflectance）设置为“constant”（固定着色）或“matte”（恒定着色）。

警告：我们的引擎只支持有一个纹理的物体，纹理的大小为 $m \times m$，其中 m 为 2 的幂，且不能大于 256。

虽然在每个物体中只能使用一个纹理，但这并不意味着不能对物体进行非常漂亮的纹理映射。只需将一个大型（如 128×128 或 256×256 的）表层作为纹理图用于物体，并指定物体的纹理坐标。前一章中的图 8.52 是一个不错的表层，它来自一个 Quake II 模型中。正如读者看到的，在同一个纹理图中，可以包含很多块；使用同一个纹理图时，只需要正确地指定纹理坐标即可。

与前一版相比，新的.COB 加载函数的很多功能相同，但新增了加载和识别纹理的功能。它从.COB 文件中读取纹理信息；基本立方体的纹理信息如下：

```
Texture Vertices 6
0.000000 0.000000
0.000000 1.000000
0.000000 0.000000
0.000000 1.000000
1.000000 0.000000
1.000000 1.000000
```

在多边形面列表中，每个顶点的第二个分量（粗体）是纹理顶点：

```
Faces 12
Face verts 3 flags 0 mat 4
<0,0> <1,1> <3,5>
Face verts 3 flags 0 mat 4
<0,0> <3,5> <2,4>
Face verts 3 flags 0 mat 0
<0,1> <2,5> <5,4>
Face verts 3 flags 0 mat 0
<0,1> <5,4> <4,0>
Face verts 3 flags 0 mat 1
<2,2> <3,3> <6,5>
Face verts 3 flags 0 mat 1
<2,2> <6,5> <5,4>
Face verts 3 flags 0 mat 0
<1,0> <7,1> <6,5>
Face verts 3 flags 0 mat 0
<1,0> <6,5> <3,4>
Face verts 3 flags 0 mat 2
<4,4> <5,0> <6,1>
Face verts 3 flags 0 mat 2
<4,4> <6,1> <7,5>
Face verts 3 flags 0 mat 3
<0,4> <4,2> <7,3>

Face verts 3 flags 0 mat 3
<0,4> <7,3> <1,5>
```

例如，立方体网格的最后一个多边形（face 11）是一个三角形，由顶点 0、7 和 1 组成，它们对应的纹理顶点分别是 4、3 和 5，相应的纹理坐标为(1.000000, 0.000000)、(0.000000, 1.000000)和(1.000000, 1.000000)。其中每对纹理坐标的格式为(u, v)：纹理图的(x, y)坐标。后面讨论纹理映射时，将更详细的介绍映射细节，现在读者只需要知道指定纹理坐标的方式即可。加载纹理坐标后，还需要加载纹理图。这有些棘手，但只要将一些细节考虑清楚，便能够正确地处理。首先，对于每个物体，只能加在一个纹理。因此，读取包含纹理图的材质时，只加载第一个纹理，而忽略其他的纹理。.COB 加载函数中加载纹理的代码如下：

```
// OBJECT4DV2 只支持一个纹理，虽然我们将加载所有的材质
// 如果是第一个纹理图，则加载它
// 并设置一个标记，禁止加载其他纹理
if (!obj->texture)
  {
  // 第 1 步：为位图分配内存
  obj->texture = (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE));

  // 根据文件名和全局绝对纹理路径加载纹理
  char filename[80];
  char path_filename[80];
  // 提取文件名
  Extract_Filename_From_Path(materials[material_index +
               num_materials].texture_file, filename);

  // 将文件名和根路径组合起来
  strcpy(path_filename, texture_path);
  strcat(path_filename, filename);

  // path_filename 现在包含文件名和根路径
  // 加载位图（8/16 位）
  Load_Bitmap_File(&bitmap16bit, path_filename);

  // 创建大小和位深合适的位图
  Create_Bitmap(obj->texture,0,0,
  bitmap16bit.bitmapinfoheader.biWidth,
  bitmap16bit.bitmapinfoheader.biHeight,
  bitmap16bit.bitmapinfoheader.biBitCount);

  // 加载位图图像
  if (obj->texture->bpp == 16)
```

```
    Load_Image_Bitmap16(obj->texture,
              &bitmap16bit,0,0,BITMAP_EXTRACT_MODE_ABS);
    else
     {
      Load_Image_Bitmap(obj->texture, &bitmap16bit,0,0,
               BITMAP_EXTRACT_MODE_ABS);
     } // end else 8 bit

   // 卸载位图
  Unload_Bitmap_File(&bitmap16bit);

  // 设置物体的属性，指出物体有纹理
  SET_BIT(obj->attr, OBJECT4DV2_ATTR_TEXTURES);

  } // end if
```

上述代码很重要，它执行了读者必须考虑的一些清理工作。首先，这些代码位于加载所有的多边形并将材质应用于每个多边形的代码之后。在.COB 文件中，使用材质索引来指定应用于多边形的材质，但材质的定义在整个模型被加载后才出现。因此，必须等待加载模型和材质后，再修改与材质相关的参数。我们的引擎并不支持使用材质，但有关纹理的信息包含在材质定义中，因此必须等到加载材质后才能知道这些信息。总之，加载材质时，如果发现它使用了纹理，将执行上述代码。

上述代码首先判断之前是否加载过纹理，如果是这样，则什么也不做；否则进入代码块。该代码块首先调用函数 Extract_Filename_From_Path()从完整的纹理图路径中提取文件名，该函数的代码如下：

```
char *Extract_Filename_From_Path(char *filepath, char *filename)
{
// 这个函数从完整的路径"../folder/.../filname.ext"中提取文件名
// 它从完整路径的最后开始向前扫描，查找第一个"\" 或"/"
// 然后将这里到末尾的内容作为文件名
// 检查路径是否有效
if (!filepath || strlen(filepath)==0)
  return(NULL);

int index_end = strlen(filepath)-1;

// 查找文件名
while( (filepath[index_end]!='\\') &&
     (filepath[index_end]!='/') &&
     (filepath[index_end] > 0) )
   index_end--;

// 将文件名复制到变量 filename 中
memcpy(filename, &filepath[index_end+1],
     strlen(filepath) - index_end);

// 返回结果
return(filename);

} // end Extract_Filename_From_Path
```

编写上述函数的原因有两个。首先，C/C++对 ANSI 字符串的支持令人失望，没有从下面这样的完整路径中提取文件名的简单方法：

```
d:\path\...\filename.ext
```

因此，需要编写一个这样的函数。该函数从完整路径的最后开始提取，直到遇到字符\或/位置。因此，像下面这样调用该函数时：

```
Extract_Filename_From_Path("d:\files\atari\object.cob", filename);
```

变量 filename 的值为“object.cob”，这正是我们需要的。这很好，但是否可以不使用完整的路径呢？如果要将纹理放在某个地方该如何办？为提供这种功能，作者决定创建一个纹理根路径，这意味着所有的纹理都必须位于根路径中。在 T3DLIB7.CPP 中，根路径被定义为：

```
char texture_path[80] = "./"; // 纹理的根路径
```

换句话说，将在被执行的.EXE 文件的工作目录下查找纹理。别忘了“./”表示当前目录。因此，所有演示程序使用的多媒体内容都必须存储在其工作目录下。读者创建自己的游戏时，可能需要创建一个MEDIA\目录，该目录包含如下子目录：

```
TEXTURES\

MODELS\
SOUNDS\
```

并将所有的纹理放在子目录 TEXTURES\中。

编写上述函数的第二个原因更为重要。请看下述.COB 文件中的材质定义：

```
Mat1 V0.06 Id 18623892 Parent 18629556 Size 00000182
mat# 0
shader: phong facet: auto32
rgb 1,0.952941,0.0235294
alpha 1 ka 0.1 ks 0.1 exp 0 ior 1
texture: 36D:\Source\models\textures\wall01.bmp
offset 0,0 repeats 1,1 flags 2
ShBx V0.03 Id 18623893 Parent 18623892 Size 00000658
Shader class: color
Shader name: "texture map" (caligari texture)
Number of parameters: 7
file name: string "D:\Source\models\textures\wall01.bmp"
S repeat: float 1
T repeat: float 1
S offset: float 0
T offset: float 0
animate: bool 0
filter: bool 0
Flags: 3
Shader class: transparency
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
Shader class: reflectance
Shader name: "phong" (phong)
Number of parameters: 5
ambient factor: float 0.1
diffuse factor: float 0.9
specular factor: float 0.1
exponent: float 3
specular colour: color (255, 255, 255)
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
END V1.00 Id 0 Parent 0 Size    0
```

从中可以发现，纹理路径是绝对路径。这意味着除非两台计算机的目录结构相同，否则在一台计算机上编写的演示程序将无法在另一台上运行。为避免这种情况发生，作者决定同时使用根目录文件名和通用的纹理路径。

回到新的.COB 加载函数。提取最终的纹理路径和名称后，将加载纹理。为此，调用加载位图的函数，创建一个纹理位图，并将位图文件的内容复制到存储位图的位图图像中。当然，这里检查了位深，并据此（8 位还是 16 位）调用相应的函数。

这里有必要讨论一下位深。有时候，可能需要使用高级包装（wrapper）函数来隐藏 8/16 位的函数，因为检查位深并据此调用相应的函数有点麻烦。这里之所以没有这样做，是因为我们知道当前使用的是 8 位模式还是 16 位模式，没有必要增加额外的函数调用，虽然通过使用函数指针或虚拟函数，可以避免这种函数调用的开销。

接下来需要讨论的是.COB 加载函数的参数 vertex_flags。通过该参数来重新指定光照模型是没有意义

的，但可能需要通过它来调整纹理坐标。在建模程序导出的.COB 文件中，纹理坐标可能并非像我们希望的那样。例如，可能需要互换或反转纹理坐标 u 和 v。为增加这种功能，定义了如下顶点标记：

```
// 反转纹理坐标 u
#define VERTEX_FLAGS_INVERT_TEXTURE_U    0x0080
//反转纹理坐标 v
#define VERTEX_FLAGS_INVERT_TEXTURE_V    0x0100
// 互换纹理坐标 u 和 v
#define VERTEX_FLAGS_INVERT_SWAP_UV      0x0800
```

设置参数 vertex_flags 时，只需使用逻辑 OR 运算将上述标记与其他标记（如 VERTEX_FLAGS_TRANSFORM_LOCAL_WORLD）组合起来即可。

既然谈到纹理坐标，这里顺便说一句，大多数建模程序导出的纹理坐标的取值范围都是 0～1。换句话说，纹理坐标被归一化。我们的光栅化函数没有采取这种方式，因此在加载函数中将纹理坐标写入纹理数组中时，必须根据纹理的大小对纹理坐标进行缩放。例如，假设被加载的纹理大小为 64×64，而纹理坐标为(0.2，0.34)，则必须使用缩放因子 63（64-1）对纹理坐标进行缩放：

(63*.2，63*.34) = (12.6，21.42)

注意：这里保留了纹理坐标的浮点数格式，这样进行纹理映射时，精度可超过像素级（sub-pixel）。

现在来总结一下新的.COB 加载函数。调用该函数的方法与以前相同。例如，下面的函数调用加载一个包含纹理的战斗机模型，反转纹理坐标 v，并考虑到了 Caligari trueSpace 使用的是右手坐标系的问题（即需要互换坐标 Y 和 Z——译者注）：

```
Load_OBJECT4DV2_COB(&obj_player,"tie02.cob",
          &vscale, &vpos, &vrot,
          VERTEX_FLAGS_INVERT_TEXTURE_V |
          VERTEX_FLAGS_SWAP_YZ |
          VERTEX_FLAGS_TRANSFORM_LOCAL_WORLD );
```

9.4 回顾多边形的光栅化

虽然前面只是粗略地讨论了引擎的基本变化，但占据的篇幅还是不短。主要原因在于引擎涉及的内容很多，也很复杂。对于讨论的每项内容，作者都将尽可能地阐述如何实现之，因此需要做的准备工作很多，请读者耐心对待。讨论 Gouraud 着色和纹理映射之前，先来复习一下多边形的光栅化。《Windows 游戏编程大师技巧》介绍过这个主题，本书前一章也简要地对此做过介绍，但在 3D 图形学领域中，多边形光栅化无处不在，99%的主题都与光栅化、插值或采样有一定的关系，因此有必要深入了解它。

作者重新编写了多边形光栅化函数，使之更为精确：而原来的光栅化函数只能使用整数坐标。另外，还定义了填充规则和其他细节，这里将讨论它们，让读者对其有所了解。

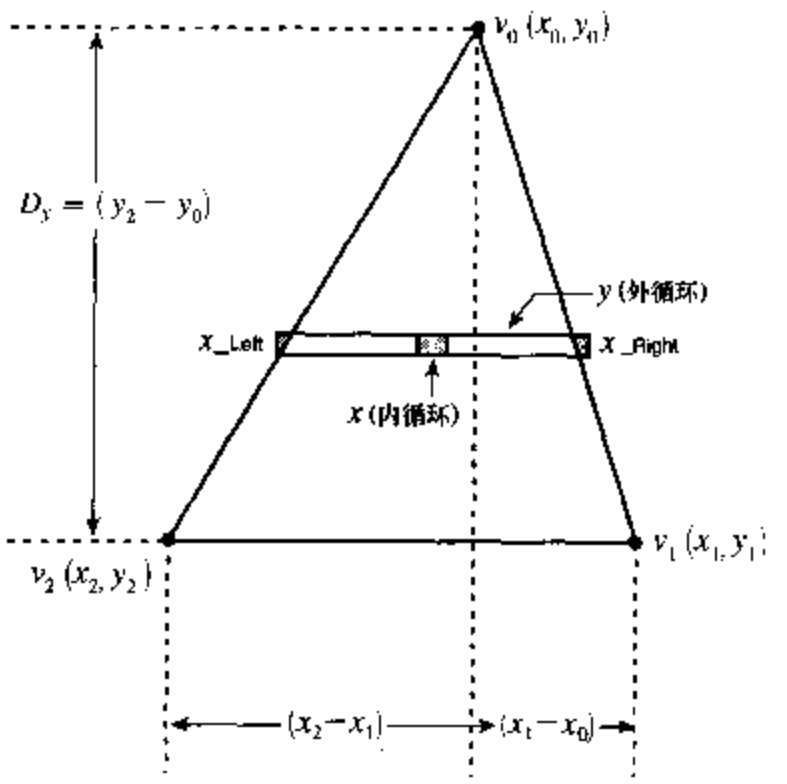


图 9.9 三角形的光栅化

9.4.1 三角形的光栅化

绘制三角形的方法很多，其中包括根据三角形的两条边确定直线的端点，然后使用直线绘制算法（如 Bresenham 算法）或简单插值来填充三角形。作者推荐使用插值算法，它更为简单明了。下面复习一下这种算法。如图 9.9 所示，只需找出三角形光栅化版本覆盖的所有点即可，图中用小圆点表示这些点。找出构成三角形的每条扫描线上各个像

素的位置后，三角形的绘制只不过是填充每个点对应的内存而已，如图 9.10 所示。

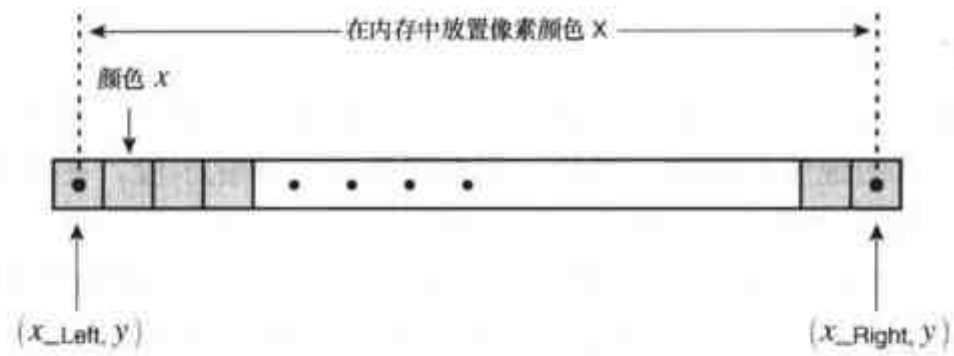


图 9.10　单条扫描线的光栅化

要找出这些点，只需要根据三角形边的斜率进行插值即可。斜率的计算方法如下：

三角形的高度为：

```
dy = (y2 - y0);
```

最上面的顶点同左下角和右下角顶点的 x 坐标差分别是：

```
dx_left_side = (x2 - x0);
dx_right_side = (x1 - x0);
```

因此，三角形左侧边的斜率为：

```
slope_left_side = dy/dx_left_side = (y2 - y0)/(x2 - x0);
```

右侧边的斜率为：

```
slope_right_side = dy/dx_right_side = (y2 - y0)/(x1 - x0);
```

然而，我们需要的并不是斜率。斜率指的是“x 坐标变化一个单位时 y 坐标的变化量”，这意味着沿 x 轴移动一个像素时，沿 y 轴的移动距离等于斜率。我们需要的并非斜率，而是斜率的倒数，即 dx/dy。这是因为我们要通过绘制扫描线来绘制三角形，即每绘制完一条扫描线，将 y 坐标加 1。也就是说 $dy = 1$，因此：

```
dx_left_side = 1 * (x2 - x0)/(y2 - y0);
dx_right_side = 1 * (x1 - x0)/(y2 - y0);
```

这就是平底三角形的绘制算法。平顶三角形的绘制算法与此类型，这里将其留给读者去完成。下面是平底三角形绘制算法的实现代码：

```
void Draw_Triangle(float x0,float y0, // 顶点 0
          float x1,float y1, // 顶点 1
          float x2,float y2, // 顶点 2
          int color)     // 颜色
{
// 这个函数用于对平底三角形进行光栅化

// 计算左侧边的梯度
float dx_left = (x2 - x0)/(y2 - y0);

// 计算右侧边的梯度
float dx_right = (x1 - x0)/(y2 - y0);

// 设置初始值
float x_left = x0;
float x_right = x0;

// 进入光栅化循环
for (int y=y0; y<=y1; y++)
   {
   // 绘制扫描线
   Draw_Line(x_left, x_right, y, color);

   // 计算下一条扫描线起点和终点的 x 坐标
   x_left+=dx_left;
   x_right+=dx_right;
```

```
    } // end for y

} // end DRAW_Triangle
```

我们的三角形光栅化函数就是以此为基础的。不过，它存在几个缺点。首先，它是基于浮点数的，这不是问题，因为计算浮点数的速度和整数一样快。但在最终的光栅化阶段，在浮点数和整数之间进行转换时将带来麻烦，稍后将讨论这一点。

真正的问题在于函数 Draw_Line()。我们假定它接受浮点数参数（可能确实是这样），但必须在某个地方将这些浮点数转换为整数，这提出了一个问题：应进行截尾（truncate）还是四舍五入；应使用函数 ceiling() 还是 floor()？这些问题都与填充规则（fill convention）相关。

提示：数学函数 ceiling(x)返回不小于 x 的最小整数，如图 9.11 所示；而函数 floor(x)返回不大于 x 的最大整数，如图 9.12 所示。

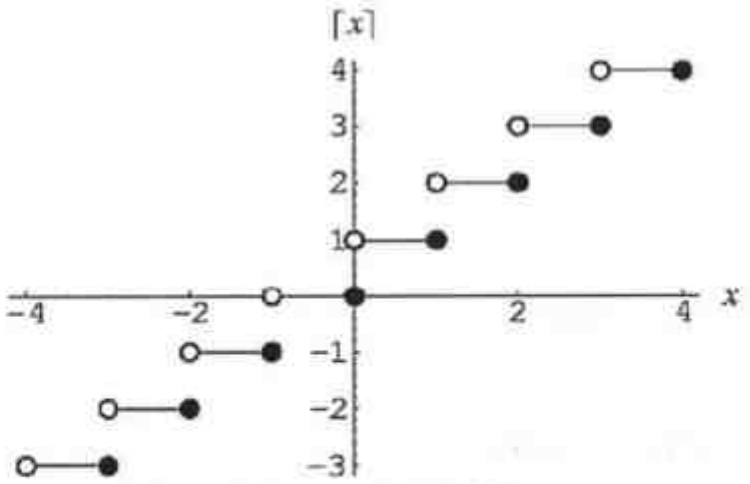


图 9.11　函数 ceiling( )的工作原理

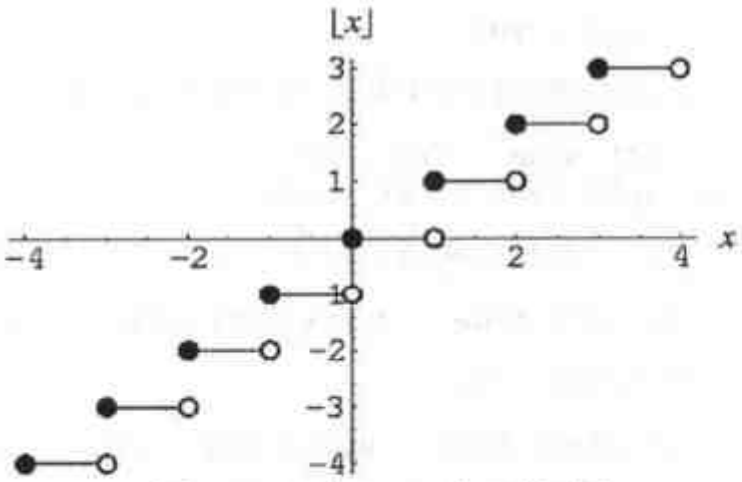


图 9.12　函数 floor( )的工作原理

填充规则指的是对多边形进行光栅化时，如何确定需要光栅化的起始像素和终止像素。这很重要，稍后将进行讨论；现在继续讨论光栅化函数的其他细节。

除了填充规则外，前述光栅化函数还做了哪些假设呢？

1．没有考虑常规三角形，需要解决这种问题。

2．没有考虑裁剪（图像空间裁剪或物体空间裁剪），这也需要解决。

首先来讨论第一点。在本书中，使用的是《Windows 游戏编程大师技巧》提供的三角形光栅化函数，这些函数将多边形分割成三种三角形，如图 9.13 所示：

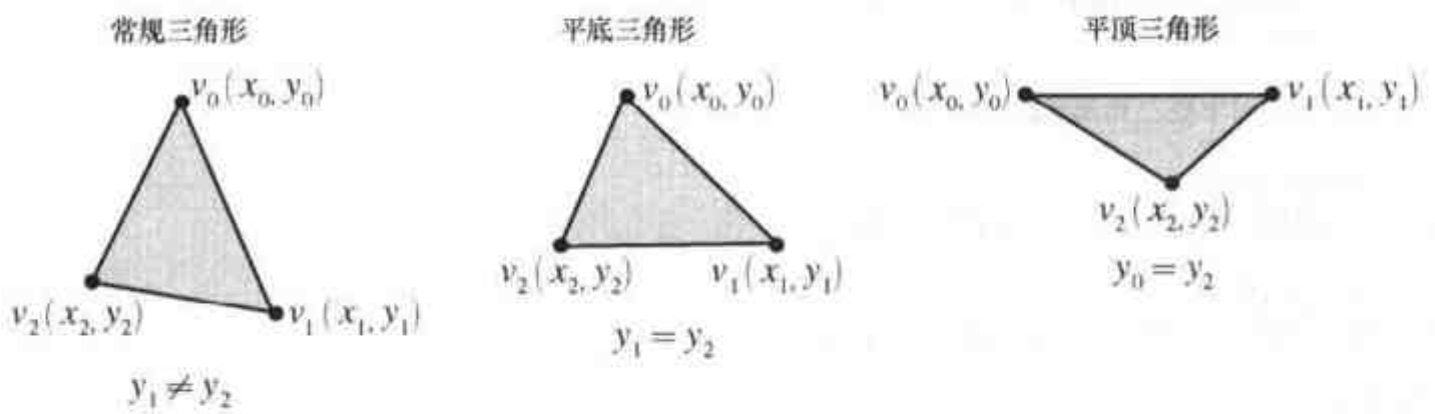


图 9.13　三角形类型

- 平顶三角形；
- 平底三角形；
- 常规三角形。

读者现在应该明白了，首先对三角形进行检测，可避免编写通用的可绘制任何三角形的光栅化函数。

绘制三角形时，有一个从 y0 到 y1 的内循环。在这个循环中，调整最左边和最右边像素的位置，然后在这两个像素之间绘制一条直线。然而，对于常规三角形，三角形左边或右边的斜率在某个时刻将发生变化。这意味着要么在 y0 到 y1 的循环中加入检测这种情况的代码，要么将常规三角形分割成两个三角形：

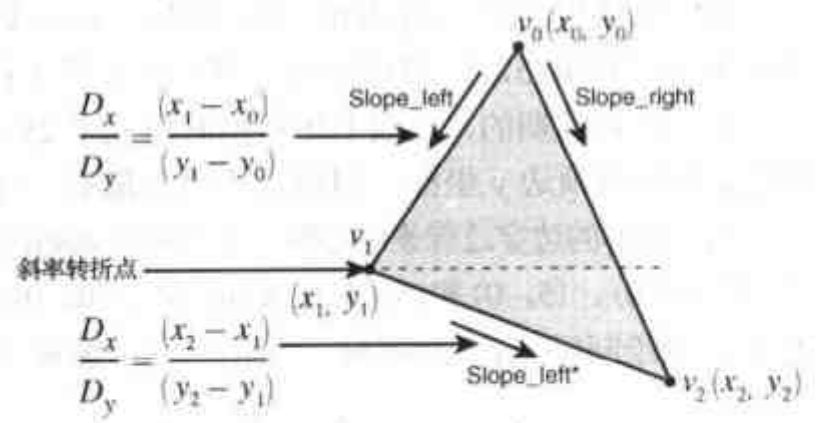
一个平底三角形和一个平顶三角形。

如果采用第二种方法，将回到绘制特殊三角形的情形。如果采用第一种方法——中途修改斜率，则光栅化如图 9.14 所示的三角形时，需要添加如下所示的条件语句：

```
for (y=y0; y < y2; y++)
  {
  Draw_Line(x_left, x_right, y, color);

  x_left+=dx_left;
  x_right+=dx_right;
// 检测斜率转折点
if (y==y1)
  dx_left = new_dx_left;

} // end for
```



基本上，在循环中需要检测是否到达斜率转折点，如果到达了，则相应地修改斜率。这提出了一个棘手的问题：在斜率转折点，x 值可能存在一定的误差。如图 9.14 所示，到达斜率转折点后，从哪里开始绘制呢？从当前扫描线开始还是下一条扫描线开始？应避免重复绘制像素（重复绘制并没有害处），更重要的是应使用准确的 y 值，即必须这样做：根据顶点（$x1$，$y1$）来确定下一条扫描线的 y 坐标。也就是说，根据变量 y 的类型（整数还是浮点数），到达斜率转折点后，不要将前一次循环时的 x 值加上 new_dx_left 来得到下一条扫描线起点的 x 坐标，而应根据顶点（$x1$，$y1$）的位置来计算它。

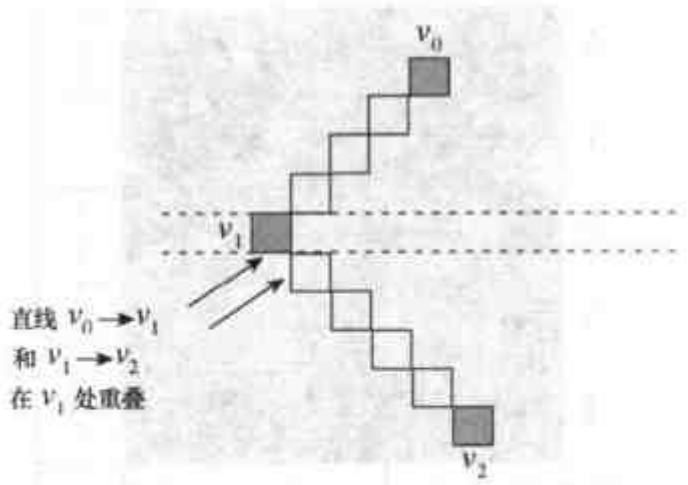


图 9.14　在三角形光栅化期间修改斜率

9.4.2　填充规则

在 3D 引擎中光栅化多边形时，通常旨在绘制实心物体的图像。这意味着多个多边形可能共用顶点。在这种情况下，需要避免重复绘制某些直线。如图 9.15a 所示，两个三角形构成了一个菱形，上面的三角形终止于第 20 行，而下面的三角形起始于第 20 行。因此，要么在绘制上面的三角形时在第 19 行终止，要

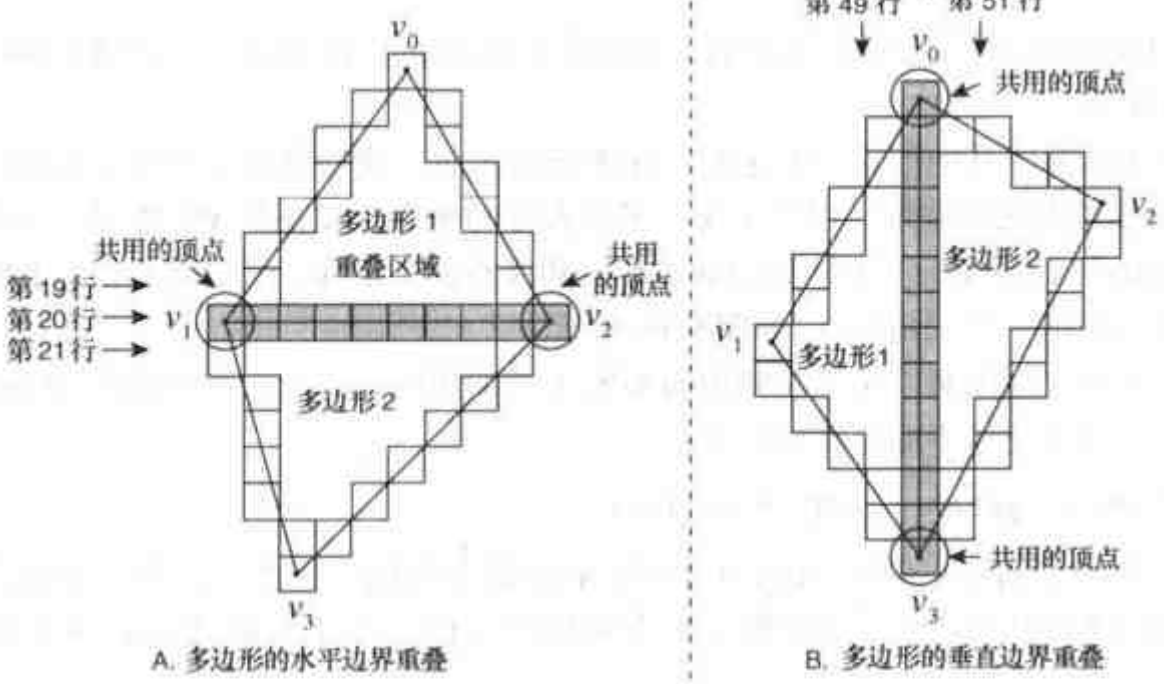


图 9.15　共用顶点和边的多边形

么绘制下面的三角形是从第 21 行开始。图 9.15b 提供了另一个例子，图中的两个三角形共用一条垂直边，它们组成了一个菱形。同样，应在绘制左边的三角形时终止于第 49 列呢？还是在绘制右边的三角形时从第 51 列开始？这些问题都与填充规则相关。

我们将使用左上（top-left）填充规则，这也是 Direct3D 在光栅化时使用的填充规则。也就是说，渲染对角为(0，0)和(5，5)的矩形时，将不填充第 5 行和第 5 列的像素，如图 9.16 所示。

正如读者预期的，绘制上述矩形时填充了 25 个像素。矩形的宽度为右边 x 坐标减左边 x 坐标，高度为底边 y 坐标减顶边 y 坐标，但最后不填充最后一行和最后一列。

当三角形的边穿过像素中心时，左上填充规则决定采取何种措施。在图 9.17 中，有两个三角形，它们的顶点分别是(0，0)、(5，0)和(5，5)以及(0，5)、(0，0)和(5，5)。绘制前一个三角形时，将填充 15 个像素（用黑色表示），而绘制后一个三角形时，将填充 10 个像素（用灰色表示），这是因为共用边为第一个三角形的左侧边。

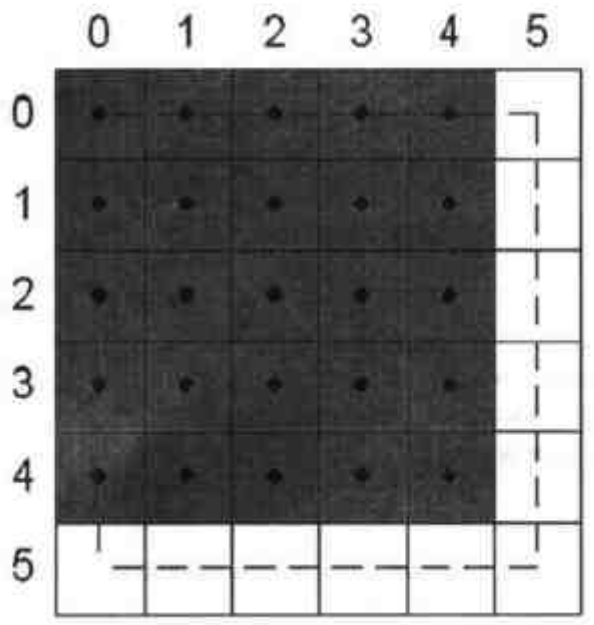


图 9.16　使用左上填充规则的矩形

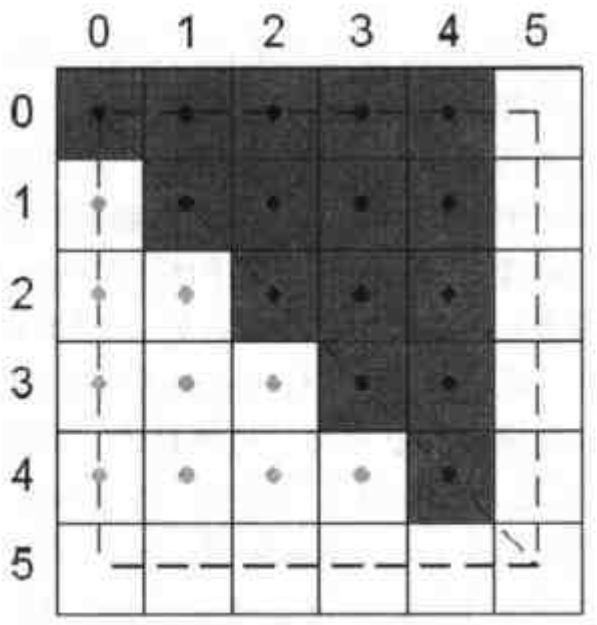


图 9.17　使用左上填充规则的矩形

我们只所以使用这种填充规则，是因为据作者所知，这是标准填充规则。为实现这种填充规则，只需使用 ceil()函数，将像素坐标调整为整数值。例如，光栅化 y 坐标范围为 $y0$ 到 $y1$ 的平底三角形时，需要这样修改这两个坐标值：

```
ystart = ceil(y0);
yend = ceil(y1) - 1;
```

在循环中计算扫描线的 x 坐标时，需要这样做：

```
xs = ceil(x_left);
xe = ceil(x_right) - 1;
```

总之，需要使用 ceil()函数计算循环的初始 y 值和终止循环的 y 值；并在绘制每条扫描线之前，使用 ceil()函数计算端点的 x 坐标。

这很容易，但还遗漏了其他因素吗？是的。在光栅化之前，我们修改了起始 y 值和终止 y 值，因此在计算 x 坐标时，需要考虑这种差别。也就是说，虽然实际的垂直跨度是从 $y0$ 到 $y1$，但使用左上填充规则时，我们根据 ceil($y0$)和 ceil($y1$) – 1 来计算扫描线起点和终点的 x 坐标，因此需要在 x_left 和 y_right 中考虑这种误差（如果绘制的是平底三角形），如图 9.18 所示。

图中说明了左上填充规则对三角形左侧边的初始 x 值的影响。因此，必须将修改 ystart 带来的影响考虑进来，这很简单，只需要像下面这样做即可：

```
x_left = x_left + (ystart - y0) * dx_left
```

换句话说，需要加上将 $y0$ 该为 ceil(y0)给 x 带来的微小变化。为此，需要计算 ceil($y0$)和 $y0$ 之间的差，然后将其乘以左侧边的梯度，以调整左侧边的初始 x 值。对于右侧边，必须使用相似的方法计算其初始 x 值。

有关光栅化三角形的最后一个细节是，光栅化之前，必须将顶点按从上到下和从左到右的顺序进行排序。

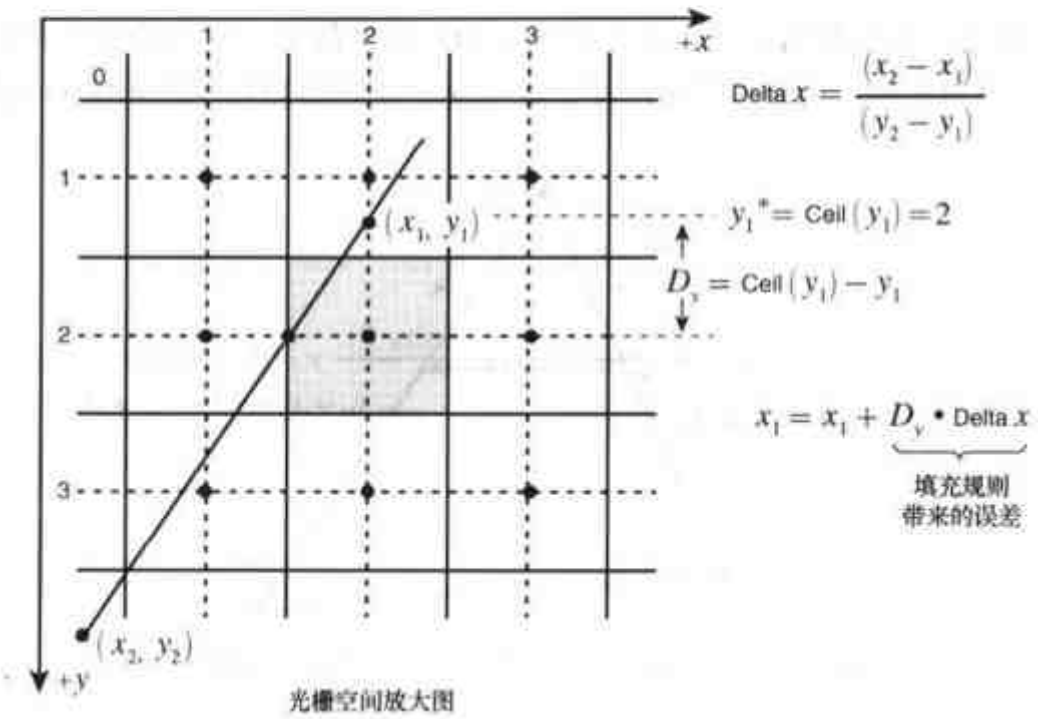


图 9.18　考虑修改 ystart 带来的影响，正确地设置 x 的值

9.4.3　裁剪

编写对三角形进行光栅化的代码之前，需要讨论的最后一个主题是裁剪。本书前面讨论过裁剪多次，这里从三角形光栅化的角度复习一下这个主题。可以根据屏幕矩形对所有三角形进行裁剪，如图 9.19 所示。然而，作者发现这样做利大于弊，一种更好的方法是，在光栅化期间进行裁剪。这很简单，只需根据裁剪类型——垂直裁剪、水平裁剪还是不裁剪——执行相应的代码块即可，如图 9.20 所示。最简单的情况是不需要裁剪，其次是垂直裁剪。当三角形延伸到屏幕顶边和/或屏幕底边之外时，需要进行垂直裁剪。

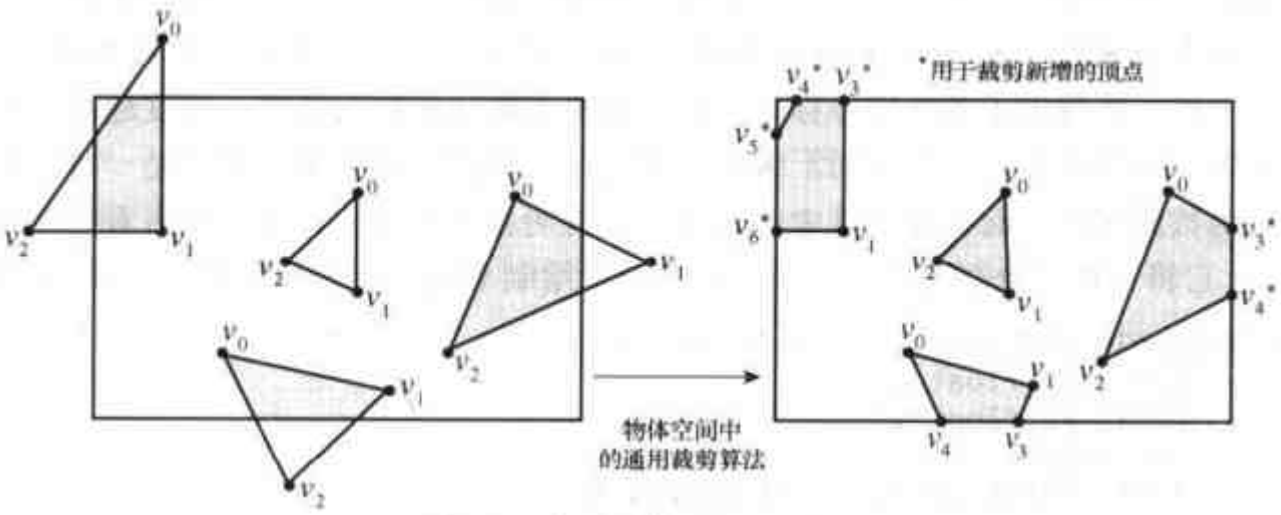


图 9.19　使用屏幕矩形进行裁剪

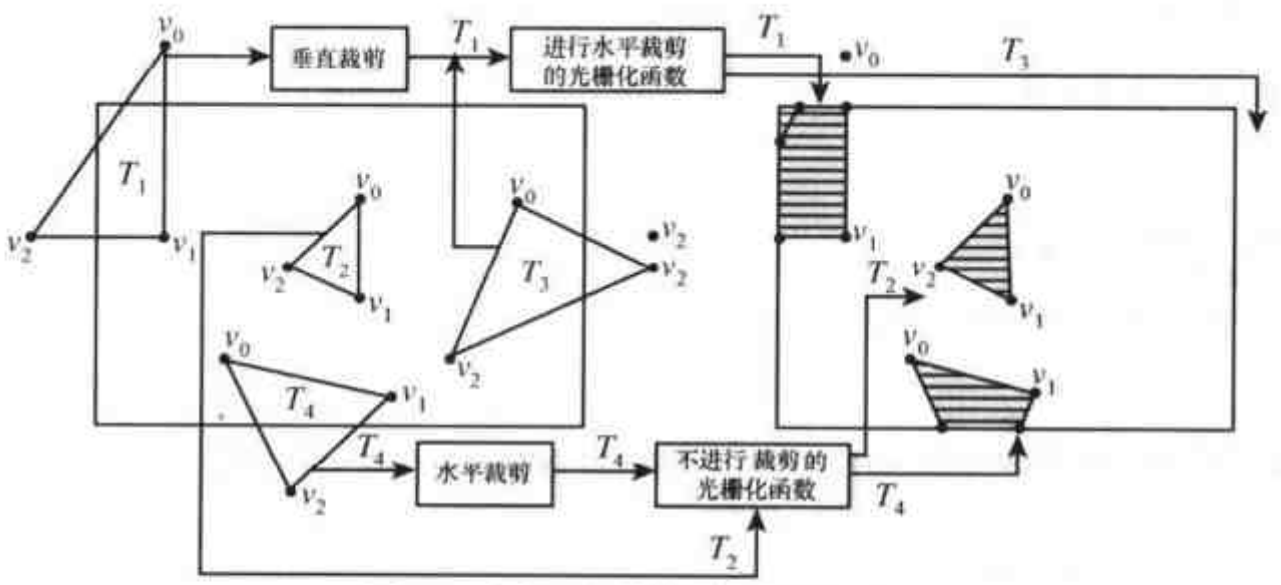


图 9.20　根据几何体使用特殊裁剪函数

例如，要根据屏幕顶边（在多数情况下，其方程为 y=0）进行裁剪，只需检查三角形最上面的顶点的 y 坐标是否小于 0。如果是，则将起始 y 值设置为 0，并计算三角形左侧边和右侧边相应的 x 值，如图 9.21 所示。

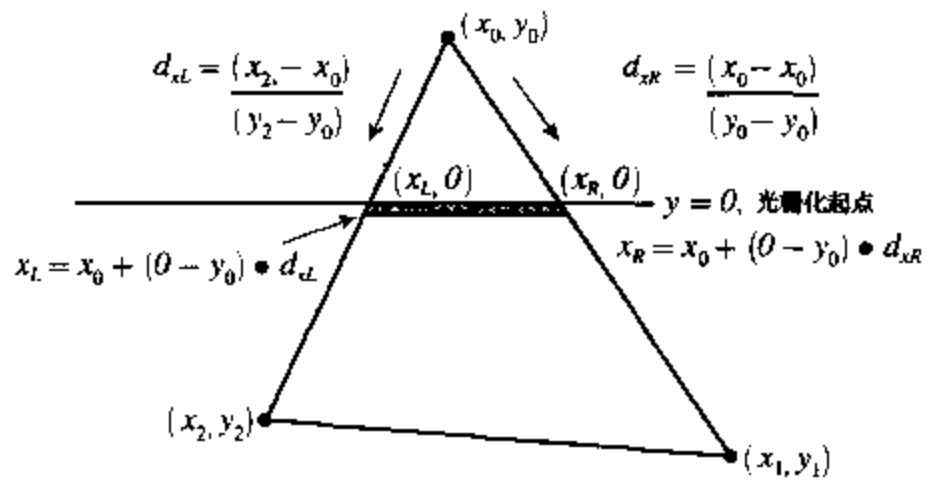


图 9.21　进行垂直裁剪后重新计算 x 坐标的起始值

裁剪三角形底部比裁剪三角形顶部容易得多：只需在到达屏幕的最后一行后停止光栅化即可。

最后一种情况是，光栅化三角形时，起点/终点的 x 坐标在窗口的左边/右边。这类似于使用屏幕顶边进行垂直裁剪：首先使用 x=0 对扫描线进行裁剪——将扫描线起点的 x 坐标设置为 0，然后右侧边的 x 执行类似的测试和裁剪；然后保存裁剪结果并绘制扫描线。

提示：稍后将执行 3D 裁剪，但那时将发现，使用视景体进行 3D 裁剪是不值得的，因为被裁剪后的三角形最多可能有 6 条边。然而，必须使用近裁剪面进行裁剪，这可能导致三角形变成四边形，需要再次将其分割成三角形。这种操作将在渲染列表上进行，被裁剪后变成四边形的三角形并不多，因此工作量不大。

9.4.4　新的三角形渲染函数

回顾了三角形光栅化的一些细节后，接下来看看新的三角形渲染函数，该函数考虑了前面讨论的各种因素。作者几乎是重新编写了 8 位和 16 位模式的三角形渲染函数，主要是因为旧版本不支持浮点数坐标，而新版本支持。另外，还提供了条件编译标记，用于指定光栅化函数的精度；精度越高，光栅化函数的速度越慢。作者认为，就现在而言，与使用汇编语言进行优化相比，使用条件编译是一种更好的选择。这里不打算列出所有函数的代码，而只介绍其中的两个，以说明新增的功能。首先来看看绘制常规三角形的函数（16 位版本），它将三角形分割为两个三角形，并调用绘制平底三角形和平顶三角形的函数来绘制它们：

```
void Draw_Triangle_2D2_16(float x1,float y1,
                          float x2,float y2,
                          float x3,float y3,
                          int color,
                          UCHAR *dest_buffer, int mempitch)
{
// 这个函数将一个三角形绘制到目标缓存中
// 它将三角形分解为一个平底三角形和一个平顶三角形

float temp_x, // 用于排序
    temp_y,
    new_x;

#ifdef DEBUG_ON
  // 记录渲染状态
  debug_polys_rendered_per_frame++;
#endif

// 判断三角形是否退化为直线
if ((FCMP(x1,x2) && FCMP(x2,x3))  ||  (FCMP(y1,y2) && FCMP(y2,y3)))
  return;

// 根据 y 坐标升序排列 p1、p2 和 p3
if (y2 < y1)
  {
```

```
   SWAP(x1,x2,temp_x);
   SWAP(y1,y2,temp_y);
   } // end if

// 此时 p1 和 p2 的排列顺序是正确的
if (y3 < y1)
  {
  SWAP(x1,x3,temp_x);
  SWAP(y1,y3,temp_y);
  } // end if

// 最后检查 y2 和 y3 哪个大
if (y3 < y2)
  {
  SWAP(x2,x3,temp_x);
  SWAP(y2,y3,temp_y);
  } // end if

// 裁剪测试
if ( y3 < min_clip_y || y1 > max_clip_y ||
  (x1 < min_clip_x && x2 < min_clip_x && x3 < min_clip_x) ||
  (x1 > max_clip_x && x2 > max_clip_x && x3 > max_clip_x) )
  return;

// 检查三角形是否是平顶的
if (FCMP(y1,y2))
  {
  Draw_Top_Tri2_16(x1,y1,x2,y2,x3,y3,color, dest_buffer, mempitch);
  } // end if
else
if (FCMP(y2,y3))
  {
  Draw_Bottom_Tri2_16(x1,y1,x2,y2,x3,y3,color, dest_buffer, mempitch);
  } // end if bottom is flat
else
  {
  // 将常规三角形分割成两个三角形
  new_x = x1 + (y2-y1)*(x3-x1)/(y3-y1);

  // 绘制分割后得到的每个三角形
  Draw_Bottom_Tri2_16(x1,y1,new_x,y2,x2,y2,color, dest_buffer, mempitch);
  Draw_Top_Tri2_16(x2,y2,new_x,y2,x3,y3,color, dest_buffer, mempitch);
  } // end else

} // end Draw_Triangle_2D2_16
```

该函数与前一版（Draw_Triangle_2D_16()）相同，惟一的差别在于，它接受的是浮点数坐标，且执行的是浮点数计算。接下来看看绘制平底三角形的函数（16位版本）：

```
void Draw_Bottom_Tri2(float x1,float y1,
             float x2,float y2,
             float x3,float y3,
             int color,
             UCHAR *dest_buffer, int mempitch)
{
// 这个函数用于绘制平底三角形

float dx_right,    // 右侧边的 dx/dy
   dx_left,   // 左侧边的 dx/dy
   xs,xe,    // 扫描线起点和终点的 x 坐标
   height,  // 三角形的高度
   temp_x,  // 排序时使用的临时变量
   temp_y,
   right,  // 用于裁剪
   left;

int iy1,iy3,loop_y;
```

```
// 将目标缓存转换为 ushort 类型
USHORT *dest_buffer = (USHORT *)_dest_buffer;

// 下一条扫描线的目标地址
USHORT *dest_addr = NULL;

// 重新计算以 16 位字为单位的 mempitch
mempitch = (mempitch >> 1);

// 检查 x2 和 x3 的顺序
if (x3 < x2)
  {
  SWAP(x2,x3,temp_x);
  } // end if swap

// 计算梯度
height = y3-y1;

dx_left  = (x2-x1)/height;
dx_right = (x3-x1)/height;

// 设置插值初始值
xs = x1;
xe = x1;

#if (RASTERIZER_MODE==RASTERIZER_ACCURATE)
// 执行垂直裁剪
if (y1 < min_clip_y)
  {
  // 计算新的 xs 和 ys
  xs = xs+dx_left*(-y1+min_clip_y);
  xe = xe+dx_right*(-y1+min_clip_y);

  // 重新设置 y1
  y1 = min_clip_y;

  // 确保遵循左上填充规则
  iy1 = y1;
  } // end if top is off screen
else
  {
  // 确保遵循左上填充规则
  iy1 = ceil(y1);

  // 调整 xs 和 xe
  xs = xs+dx_left*(iy1-y1);
  xe = xe+dx_right*(iy1-y1);
  } // end else

if (y3 > max_clip_y)
  {
  // 垂直裁剪
  y3 = max_clip_y;

  // 确保遵循左上填充规则
  iy3 = y3-1;
  } // end if
else
  {
  // 确保遵循左上填充规则
  iy3 = ceil(y3)-1;
  } // end else
#endif

#if ( (RASTERIZER_MODE==RASTERIZER_FAST) ||
   (RASTERIZER_MODE==RASTERIZER_FASTEST) )
```

```
// 执行垂直裁剪
if (y1 < min_clip_y)
  {
  // 计算新的 xs 和 ys
  xs = xs+dx_left*(-y1+min_clip_y);
  xe = xe+dx_right*(-y1+min_clip_y);

  // 重新设置 y1
  y1 = min_clip_y;
  } // end if top is off screen

if (y3 > max_clip_y)
  y3 = max_clip_y;

// 确保遵循左上填充规则
iy1 = ceil(y1);
iy3 = ceil(y3)-1;
#endif

// 计算视频缓存的起始地址
dest_addr = dest_buffer+iy1*mempitch;

// 检查是否需要执行水平裁剪
if (x1 >= min_clip_x && x1 <= max_clip_x &&
  x2 >= min_clip_x && x2 <= max_clip_x &&
  x3 >= min_clip_x && x3 <= max_clip_x)
  {
  // 绘制三角形
  for (loop_y = iy1; loop_y <= iy3; loop_y++,dest_addr+=mempitch)
    {
    // 绘制扫描线
    Mem_Set_WORD(dest_addr+(unsigned int)(xs), color,
          (unsigned int)((int)xe-(int)xs+1));
    // 调整扫描线起点和终点的 x 坐标
    xs+=dx_left;
    xe+=dx_right;
    } // end for

  } // end if no x clipping needed
else
  {
  // 执行水平裁剪

  // 绘制三角形

  for (loop_y = iy1; loop_y <= iy3; loop_y++,dest_addr+=mempitch)
    {
    // 裁剪
    left  = xs;
    right = xe;

    // 调整扫描线起点和终点
    xs+=dx_left;
    xe+=dx_right;

    // 对扫描线进行裁剪
    if (left < min_clip_x)
    {
    left = min_clip_x;

    if (right < min_clip_x)
      continue;
    }

    if (right > max_clip_x)
     {
```

```
        right = max_clip_x;

        if (left > max_clip_x)
          continue;
        }
        // 绘制扫描线
        Mem_Set_WORD(dest_addr+(unsigned int)left, color,
              (unsigned int)((int)right-(int)left+1));
        } // end for
      } // end else x clipping needed
    } // end Draw_Bottom_Tri2_16
```

同样，这里也使用了这样的命名规则：在主函数名后加上 2，但接口保持不变。该函数支持填充规则和条件编译标记（代码中用粗体表示），这些标记的定义如下：

```
// 这些是为新的光栅化函数定义的条件编译常量
// 为避免编写过多的函数，在函数中根据条件来改变光栅化逻辑
#define RASTERIZER_ACCURATE 0 // 遵循填充规则，精度达到像素级
#define RASTERIZER_FAST   1
#define RASTERIZER_FASTEST  2

// 指定要求的精度
#define RASTERIZER_MODE    RASTERIZER_ACCURATE
```

要修改条件编译，只需在 T3DLIB7.H 文件中修改上述代码中以粗体显示的内容即可。例如，将 RASTERIZER_ACCURATE 改为 RASTERIZER_FAST，可以提高光栅化函数的速度，但其代价是精度将降低。这个函数实现了前面讨论过的各项特性，渲染平顶三角形的函数与此类似。另外，还有 8 位模式的函数版本，在本章前面函数清单中列出了它们。

接下来简要地讨论一些可对这些光栅化函数进行的优化。

9.4.5 优化

这些光栅化函数的优点是，它们都很小，可优化的空间有限。那么哪些地方需要优化呢？算法是合理的，可供修改的地方不多。光栅化不像排序那样通过使用新的算法可极大地提高效率。另外，内部循环中的代码也很少：只是执行两三次加法以及绘制一条直线而已。惟一可优化的地方是，在数据转换中执行了一些不需要的运算。请看下述代码片段，它是渲染函数中的内循环：

```
// 绘制三角形
for (loop_y = iy1; loop_y <= iy3; loop_y++, dest_addr+=mempitch)
  {
  // 绘制扫描线
  Mem_Set_WORD(dest_addr+(unsigned int) (xs),
        color,(unsigned int) ((int)xe-(int)xs+1));

  // 调整扫描线起点和终点
  xs+=dx_left;
  xe+=dx_right;
  } // end for
```

惟一有待改进的是用粗体表示的代码行，它调用函数 Mem_Set_WORD()。这是一个内联函数，但仍有函数调用开销，可以直接使用内联的 ASM 代码来实现 WORD 移动功能。另一个让作者苦恼的问题是类型转换。作者只在确定是否绘制整条扫描线时遵循了填充规则，在绘制每条扫描线时没有这样做，因为没有对扫描线端点的 x 坐标调用 ceil()函数，因此有些像素将被重复绘制。调用 ceil()函数的开销很大。要遵循填充规则，必须使用速度更快的 ceil()函数，并分析如何执行到整数的转换。这需要使用汇编语言，因为高级语言编译器过于粗造，使用高级语言无法编写出严密的内循环代码。有关使用汇编语言进行优化，将在第 16 章介绍。

这里简要地讨论一下函数 ceil()和 floor()。这两个函数位于标准 C\C++库中，在多边形光栅化函数的精确版本中，需要每个多边形调用了这两个函数，调用次数可能高达 10～20 000 次，因此有必要提高它们的速度。为此，方法之一是根据参数的可能取值范围编写自己的 ceil()函数。换句话说，我们知道要渲染的三角形的坐标范围位于(0，screen_width)和(0，screen_height)内，可以据此为 ceil()和 floor()创建一个查找表。

再次来看看函数 ceil()的代码：

```
ceil(x) = { if x >= 0:
            if x=int(x) then ceil(x) = x
            else x=int(x)+1

          { if x < 0: ceil(x) = int(x)
```

归根结底，计算 ceil(x)时，决定性因素是 x 是否包含小数部分。如果是，则结果为下一个整数；否则结果为 x（x>=0 时）。如果 x<0，只需截除小数部分即可。下面是一种算法：

```
inline int Ceiling(float x)
{
if (x < 0) return (int)x;
else
if ((int)x < x) return((int)x+1);

} // end if
```

如果使用内联的浮点数 ASM，只需几条指令便可实现上述算法。函数 floor()的情况与此类似。这里提供了优化这些数学函数的一些思路。

这里说这么多只是想提醒读者注意这样一个事实：不管浮点数处理器和整数处理器的速度有多快，在最终计算整型像素位置时，都需要付出进行类型转换的代价。这也是有时候采用定点数运算更合理的原因：使用一种数据类型（int）来执行所有的运算，并在光栅化时将前 16 位作为像素的位置。在进行 Gouraud 着色和仿射纹理映射时，作者使用的就是这种方法。

最后，应该将优化的重点放在内部循环中，正如前面指出的，在每一帧中，这些代码可能调用 10～20 000 次。因此，对于那些需要对每个多边形执行的操作，如果能够避免将意义重大。所以，将三角形分割成平底三角形和平顶三角形，然后调用相应的函数来绘制它们的做法有待于优化，后面将采用对于每个多边形调用一个绘制函数的做法。

在编写 Gouraud 着色函数和仿射纹理映射函数时，作者采取的也是这种策略——渲染函数处理常规三角形，不对其进行分割。

光栅化三角形的方法很多，您总是能够进一步提高其速度。如果按这里介绍的方法使用汇编语言、SIMD 等进行优化，可将速度提高大约 2～10 倍。

9.5　实现 Gouraud 着色处理

本章的主题是 Gouraud 着色和仿射纹理映射，但前面的准备工作必不可少。图 9.22 是一个引擎运行时的屏幕

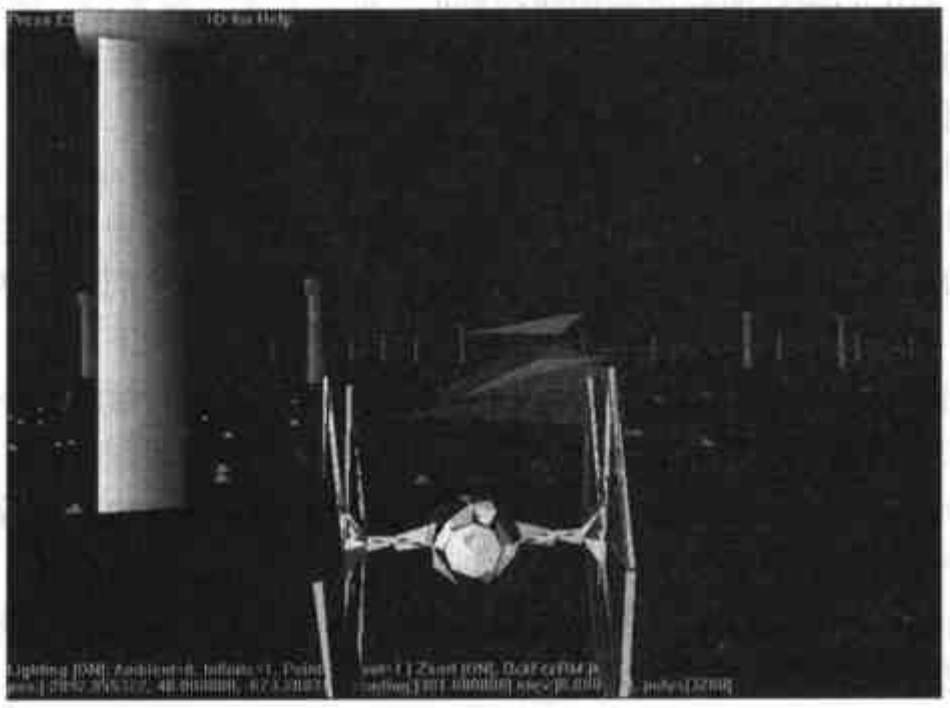


图 9.22　执行 Gouraud 着色和纹理映射的 3D 引擎

截图，请看其中照射到纹理上的彩色光线和采用平滑着色的多边形，是不是非常棒？下面介绍这是如何实现的。

正如前一章讨论光照时指出的，Gouraud 着色和恒定着色之间的区别在于，它计算每个顶点处的光照强度，而不是根据每个多边形的法线计算整个多边形的光照强度；在光栅化阶段，将通过插值计算三角形中各点的光照强度。

图 9.23 说明了插值过程。具体地说，根据每个顶点的法线（而不是多边形法线）执行光照计算，然后计算入射光强度的 RGB 值，并将其作为顶点颜色存储在顶点中。然后，使用线性插值计算三角形中各点的颜色，这样便得到了一个使用 Gouraud 着色的三角形。

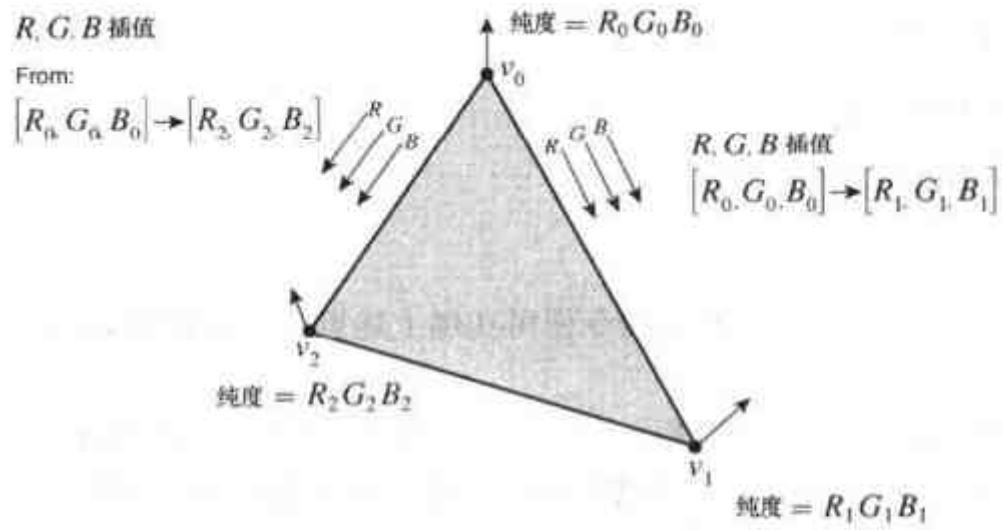


图 9.23　多边形中的 Gouraud 插值

接下来编写 Gouraud 着色引擎，它由两部分组成：

- 第 1 部分——需要一个根据面法线执行光照计算的光照引擎，我们有这样的引擎。因此，只需对每个使用 Gouraud 着色的多边形，根据其每个顶点的法线计算其光照强度。
- 第 2 部分——需要编写一个 Gouraud 光栅化函数，它根据多边行每个顶点的颜色，通过插值计算多边形中其他各点的颜色。

第 1 部分的工作几乎已经完成，我们暂时假设将 OBJECT4DV2 或 RENDERLIST4DV2 传递给光照模块时，它将像以前那样对多边形进行光照处理：执行固定着色处理、恒定着色，现在还有 Gouraud 着色。如果多边形被设置为使用 Gouraud 着色，则经过光照处理后的顶点颜色将存储在 lit_color[0]、lit_color[1]和 lit_color[2]中；如果被设置为使用固定着色和恒定着色，则整个多边形的颜色将存储在 lit_color[0]中。在做出上述假设的前提下，问题将非常简单：只需编写一个整个的三角形光栅化函数，即它将三角形作为一个整体进行光栅化（不对其进行分割——译者注），同时根据顶点的颜色通过插值计算每个像素的颜色。

9.5.1　没有光照时的 Gouraud 着色

本节标题的意思是，不关心顶点的颜色是如何得到的，可以是手工设置的，使用光照引擎计算得到的，从静态场景中查找得到的，等等。关键之处在于，渲染三角形时，只需根据顶点的颜色通过插值计算出各个像素的颜色。这个问题很简单，之前已经多次介绍过插值。

图 9.24 说明了如何实现 Gouraud 插值。有一个常规三角形，其顶点为 *v*0、*v*1 和 *v*2，它们的排列顺序如图中所示。另外，假设每个顶点都有一个 8.8.8 格式的 RGB 颜色。那么如何对该三角形进行光栅化呢？我们计算三角形左侧边和右侧边的梯度，用变量 dx_left 和 dx_right 存储它们。这些梯度表示在左侧边和右侧边上，*y* 坐标变化一个单位时 *x* 坐标的变化量，因此从一条扫描线移到下一条扫描线时，只需将扫描线起点和终点的 *x* 坐标分别加上 dx_left 和 dx_right，然后在起点和终点之间绘制一条直线。使用 Gouraud 着色时，还需要通过插值来计算扫描线端点的颜色，它们是 *y* 的函数。下面假设颜色为强度，并使用伪代码来完成这项工作。颜色为 RGB 值时与此类似，只是需要执行三次计算，每次计算一个分量。现在暂时不考虑填充规则、顶点排序等问题，以简化插值工作。

这里先介绍手工插值过程，以便读者了解其原理。在最后的函数中，使用的变量名稍有不同；这里使用的变量名只是为了方便讨论。请看图 9.24，其中的常规三角形由顶点 v0、*v*1 和 *v*2 构成，它们按顺时针方向排列。另外，还有梯度和坐标位置等。下面介绍渲染该三角形的步骤。

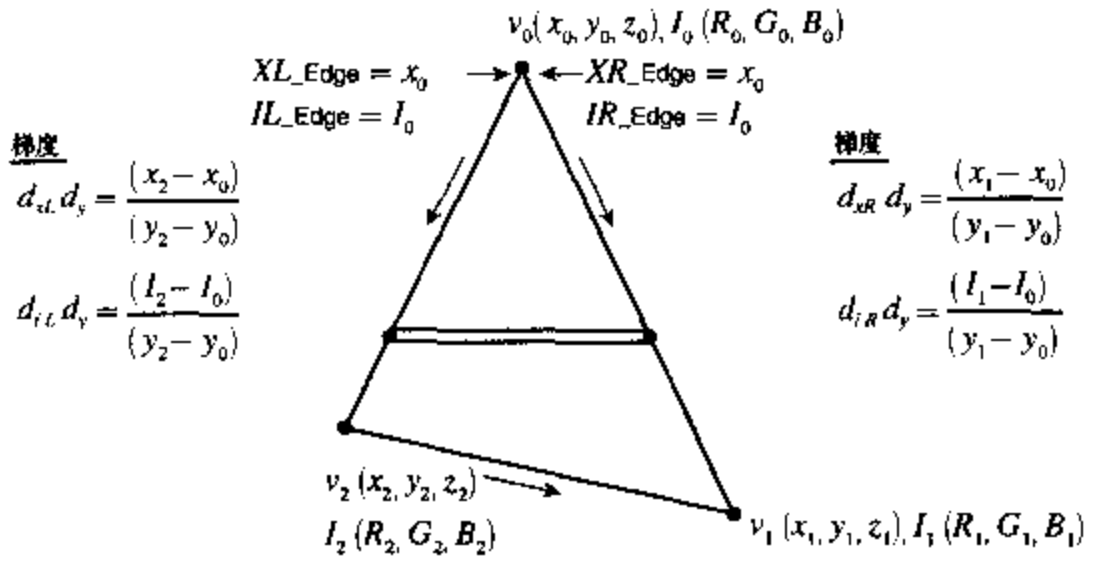


图 9.24　Gouraud 插值

第 1 步　初始化上述设置，并计算左侧边和右侧边的梯度。

下面的变量用于存储三角形左侧边和右侧边上当前点的 x 坐标：

```
xl_edge = x0
xr_edge = x0
```

下面的变量用于存储三角形左侧边和右侧边上当前点的位置颜色强度：

```
il_edge = i0
ir_edge = i0
```

第 2 步　计算左侧边和右侧边上的各种梯度。

下面是左侧边上 x 坐标梯度：

```
dxldy = (x2 - x0)/(y2 - y0)
```

右侧边上 x 坐标梯度与此类似：

```
dxrdy = (x1 - x0)/(y1 - y0)
```

下面计算颜色强度梯度，它们是 y 坐标的函数：

```
dildy = (i2 - i0)/(y2 - y0)
```

右侧边上的颜色梯度与此类似：

```
dirdy = (i1 - i0)/(y1 - y0)
```

第 3 步　进入 y 坐标变化循环，y 坐标从 $y0$ 递增到 $y2$（对应于三角形的上半部分）。

第 4 步　水平光栅化步骤。此时，需要绘制一条扫描线，但由于需要通过水平插值来计算颜色，因此每次只能绘制一个像素。

设置扫描线起点的 x 坐标和强度：

```
i = il_edge
x = xl_edge
```

计算水平强度梯度，它是 x 坐标的函数：

```
dix = (ir_edge - il_edge) / (xr_edge - xl_edge)
```

第 5 步　通过插值计算每个像素的颜色，并使用计算得到的颜色绘制该像素：

```
For (x = xl_edge to xr_edge)
   begin
   setpixel(x,y,i)
   // 更新颜色强度
   i = i + dix
   end
```

第 6 步　进入下一条扫描线。

首先计算扫描线起点和终点的 x 坐标：

```
xl_edge = xl_edge + dxldy
xr_edge = xr_edge + dyrdy
```

然后，计算扫描线起点和终点的颜色强度：

```
il_edge = il_edge + dildy
ir_edge = ir_edge + dirdy
```

最后，重复第 4～5 步。

第 7 步 使用相同的方法处理三角形的下半部分，此时需要更新左侧边的梯度。

请花些时间阅读上述伪代码，直到完全理解它们为止。简单地说，除了需要计算扫描线起点和终点的强度外，这里的光栅化与标准三角形光栅化相同。不是使用相同的颜色来绘制每条扫描线，而是根据扫描线端点的强度，通常插值来计算扫描线上每个像素的强度。下面将上述伪代码转换为粗略的 C\C++代码，以方便读者纵览整个算法：

```
void Draw_Gouraud_Triangle(float x0, float y0, float i0, // 顶点 0
                           float x0, float y0, float i0, // 顶点 1
                           float x0, float y0, float i0, // 顶点 2
{
// 假设顶点按图 9.24 所示的顺序排列

// 初始化起始位置
xl_edge = x0;
xr_edge = x0;


// 初始化颜色强度
il_edge = i0;
ir_edge = i0;

// 计算各种梯度
dxldy = (x2 - x0)/(y2 - y0);
dxrdy = (x1 - x0)/(y1 - y0);
dildy = (i2 - i0)/(y2 - y0);
dildy = (i1 - i0)/(y1 - y0);

// 进入 y 循环（三角形的上半部分）
for (y=y0; y < y2; y++)
  {
  // 设置 x 坐标和颜色强度的起始值
  i = il_edge
  x = xl_edge

  // 计算强度的水平梯度，它是 x 坐标的函数
  dix = (ir_edge - il_edge) / (xr_edge - xl_edge);

  // 通过插值光栅化扫描线的每个像素

  for (x = xl_edge; x < xr_edge; x++)
    {
    // 在(x,y)处使用颜色强度 i 绘制一个像素
    setpixel(x,y,i);

    // 更新颜色强度
    i = i + dix
    } // end for


  // 计算下一条扫描线起点和终点的 x 坐标和颜色强度
  xl_edge = xl_edge + dxldy;
  xr_edge = xr_edge + dyrdy;
  il_edge = il_edge + dildy;
  ir_edge = ir_edge + dirdy;

  } // end for y
```

```
// 在这里加上绘制三角形下半部分的代码
// 只需修改左侧边的梯度，并执行与上面相同的逻辑

} // end Draw_Gouraud_Triangle
```

总之，前面的伪代码和小型 C\C++函数说明了渲染 Gouraud 着色三角形（根据各个顶点的颜色或强度，通过水平和垂直平滑插值来光栅化三角形）的过程。那么，实际编程时该如何做呢？坦率地说，需要通过插值计算红、绿、蓝分量，而不是单个强度值。对于每条扫描线，按从左到右的顺序，通过插值计算每个像素的 R、G、B 值，然后根据结果绘制该像素，如图 9.25 所示。

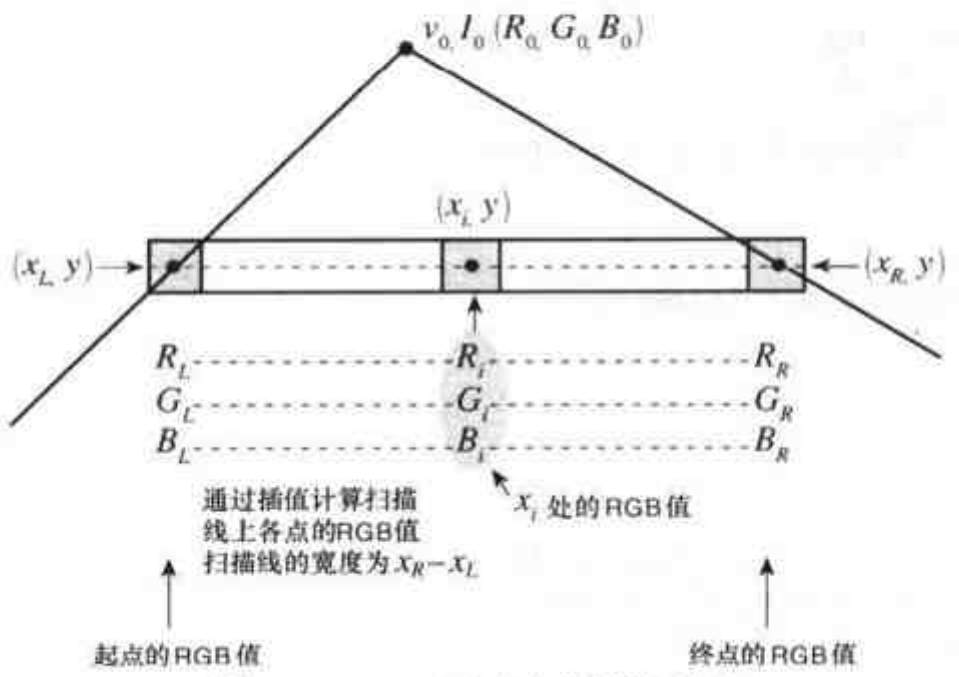


图 9.25　Gouraud 着色中光栅化扫描线

对于这种插值，惟一需要注意的是速度。确定每条扫描线的端点时，需要通常插值计算 6 个颜色分量和两个坐标分量；另外，绘制每条扫描线时，需要通过插值从左到右依次计算每个像素的 RGB 颜色，这涉及 3 个分量，如图 9.25 所示。因此，插值的速度必须足够快。

作者经过两次尝试后，决定使用 16.16 格式的定点数计算。这样做并非因为定点数计算的速度比浮点数快，而是想在绘制像素和计算 RGB 字（word）时避免执行浮点数到整数转换，因为编译器执行这种转换时使用的代码效率不高。虽然这个函数很长，这里还是决定列出其代码，因为它很复杂。为充分利用奔腾处理器的流水线技术（pipelining），采取了类似于 RISC 的方式编写这些代码；为最大限度实现并行性，代码较为冗长。整个函数的代码有 30 页左右，这里没有全部列出，而只列出处理第一种情形的代码，但其中包含读者需要明白的所有内容：

```
void Draw_Gouraud_Triangle16(POLYF4DV2_PTR face,  // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch)      // 每行占据的字节数（320、640 等）
{
// 这个函数绘制使用 Gouraud 着色的多边形
// 它是在仿射纹理映射函数的基础上修改而成的，但通过插值计算 R、G、B 值，而不是纹理坐标
// 我们通过插值计算多边形中各个像素的 R、G、B 值
// 因此 u 表示 R 值，v 表示 G 值，w 表示 B 值

int v0=0,
  v1=1,
  v2=2,
  temp=0,
  tri_type = TRI_TYPE_NONE,
  irestart = INTERP_LHS;

int dx,dy,dyl,dyr,    // 存储差值
  u,v,w,
  du,dv,dw,
  xi,yi,       // 当前的 x 和 y 坐标
  ui,vi,wi,      // 当前的 R、G、B 值
  index_x,index_y,  // 循环变量
```

```
    x,y,        // 存储一般性 x 和 y 坐标
    xstart, xend, ystart, yrestart, yend, xl,
    dxdyl, xr, dxdyr, dudyl, ul, dvdyl, vl, dwdyl,
    wl, dudyr, ur, dvdyr, vr, dwdyr, wr;

  int x0,y0,tu0,tv0,tw0,  // 顶点的初始坐标和颜色值
    x1,y1,tu1,tv1,tw1,
    x2,y2,tu2,tv2,tw2;

  int r_base0, g_base0, b_base0, // 顶点的初始颜色
    r_base1, g_base1, b_base1,
    r_base2, g_base2, b_base2;

  USHORT *screen_ptr = NULL,
      *screen_line = NULL,
      *textmap   = NULL,
      *dest_buffer = (USHORT *)_dest_buffer;

  #ifdef DEBUG_ON
    // 记录渲染状态
    debug_polys_rendered_per_frame++;
  #endif

  // 将内存跨距的单位调整为字（除以 2）
  mem_pitch >>=1;

  // 判断三角形是否在屏幕内
  if (((face->tvlist[0].y < min_clip_y) &&
     (face->tvlist[1].y < min_clip_y) &&
     (face->tvlist[2].y < min_clip_y)) ||

    ((face->tvlist[0].y > max_clip_y) &&
     (face->tvlist[1].y > max_clip_y) &&
     (face->tvlist[2].y > max_clip_y)) ||

    ((face->tvlist[0].x < min_clip_x) &&
     (face->tvlist[1].x < min_clip_x) &&
     (face->tvlist[2].x < min_clip_x)) ||

    ((face->tvlist[0].x > max_clip_x) &&
     (face->tvlist[1].x > max_clip_x) &&
     (face->tvlist[2].x > max_clip_x)))
    return;

  // 判断三角形是否退化为直线
  if ( ((face->tvlist[0].x==face->tvlist[1].x) &&
     (face->tvlist[1].x==face->tvlist[2].x)) ||
     ((face->tvlist[0].y==face->tvlist[1].y) &&
       (face->tvlist[1].y==face->tvlist[2].y)))
    return;

  // 对顶点进行排序
  if (face->tvlist[v1].y < face->tvlist[v0].y)
    {SWAP(v0,v1,temp);}

  if (face->tvlist[v2].y < face->tvlist[v0].y)
    {SWAP(v0,v2,temp);}

  if (face->tvlist[v2].y < face->tvlist[v1].y)
    {SWAP(v1,v2,temp);}

  // 判断三角形是否是平顶的
  if (face->tvlist[v0].y==face->tvlist[v1].y)
    {
    // 设置三角形类型
    tri_type = TRI_TYPE_FLAT_TOP;
```

```
  // 将顶点按从左到右的顺序排列
  if (face->tvlist[v1].x < face->tvlist[v0].x)
    {SWAP(v0,v1,temp);}
  } // end if
else
// 判断三角形是否是平底的
if (face->tvlist[v1].y==face->tvlist[v2].y)
  {
  // 设置三角形类型
  tri_type = TRI_TYPE_FLAT_BOTTOM;
  // 将顶点按从左到右的顺序排列
  if (face->tvlist[v2].x < face->tvlist[v1].x)
    {SWAP(v1,v2,temp);}
  } // end if
else
  {
  // 肯定是常规三角形
  tri_type = TRI_TYPE_GENERAL;
  } // end else

// 假定格式为 5.6.5 的
// 不能在内循环中调用函数（那样开销太大）
// 因此如果要支持 5.6.5 和 5.5.5 格式，必须编写两个硬编码版本
_RGB565FROM16BIT(face->lit_color[v0], &r_base0, &g_base0, &b_base0);
_RGB565FROM16BIT(face->lit_color[v1], &r_base1, &g_base1, &b_base1);
_RGB565FROM16BIT(face->lit_color[v2], &r_base2, &g_base2, &b_base2);
// 扩展到 8 位
r_base0 <<= 3; g_base0 <<= 2; b_base0 <<= 3;
//扩展到 8 位
r_base1 <<= 3; g_base1 <<= 2; b_base1 <<= 3;
//扩展到 8 位
r_base2 <<= 3; g_base2 <<= 2; b_base2 <<= 3;

// 提取各个顶点的坐标值和 R、B、B 值
x0 = (int)(face->tvlist[v0].x+0.5);
y0 = (int)(face->tvlist[v0].y+0.5);

tu0 = r_base0; tv0 = g_base0; tw0 = b_base0;
x1 = (int)(face->tvlist[v1].x+0.5);
y1 = (int)(face->tvlist[v1].y+0.5);

tu1 = r_base1; tv1 = g_base1; tw1 = b_base1;

x2 = (int)(face->tvlist[v2].x+0.5);
y2 = (int)(face->tvlist[v2].y+0.5);

tu2 = r_base2; tv2 = g_base2; tw2 = b_base2;

// 设置斜率转折点
yrestart = y1;

// 判断三角形类型
if (tri_type & TRI_TYPE_FLAT_MASK)
  {
  if (tri_type == TRI_TYPE_FLAT_TOP)
  {
  // 计算各种差值
  dy = (y2 - y0);
  dxdyl = ((x2 - x0)  << FIXP16_SHIFT)/dy;
  dudyl = ((tu2 - tu0) << FIXP16_SHIFT)/dy;
  dvdyl = ((tv2 - tv0) << FIXP16_SHIFT)/dy;
  dwdyl = ((tw2 - tw0) << FIXP16_SHIFT)/dy;
  dxdyr = ((x2 - x1)  << FIXP16_SHIFT)/dy;
  dudyr = ((tu2 - tu1) << FIXP16_SHIFT)/dy;
  dvdyr = ((tv2 - tv1) << FIXP16_SHIFT)/dy;
  dwdyr = ((tw2 - tw1) << FIXP16_SHIFT)/dy;
```

```
    // 垂直裁剪测试
    if (y0 < min_clip_y)
      {
      // 重新计算 y 坐标差值
      dy = (min_clip_y - y0);
      // 计算第一条扫描线起点的各种值
      xl = dxdyl*dy + (x0 << FIXP16_SHIFT);
       ul = dudyl*dy + (tu0 << FIXP16_SHIFT);
      vl = dvdyl*dy + (tv0 << FIXP16_SHIFT);
      wl = dwdyl*dy + (tw0 << FIXP16_SHIFT);
      // 计算第一条扫描线终点的各种值
      xr = dxdyr*dy + (x1 << FIXP16_SHIFT);
      ur = dudyr*dy + (tu1 << FIXP16_SHIFT);
      vr = dvdyr*dy + (tv1 << FIXP16_SHIFT);
      wr = dwdyr*dy + (tw1 << FIXP16_SHIFT);
      // 计算第一条扫描线的 y 坐标
      ystart = min_clip_y;
      } // end if
    else
     {
     // 不用裁剪
     // 设置第一条扫描线起点和终点的各种值
     xl = (x0 << FIXP16_SHIFT);
     xr = (x1 << FIXP16_SHIFT);
     ul = (tu0 << FIXP16_SHIFT);
     vl = (tv0 << FIXP16_SHIFT);
     wl = (tw0 << FIXP16_SHIFT);
     ur = (tu1 << FIXP16_SHIFT);
     vr = (tv1 << FIXP16_SHIFT);
     wr = (tw1 << FIXP16_SHIFT);
     // 设置第一条扫描线的 y 坐标
     ystart = y0;
     } // end else
    } // end if flat top
    else
    {
    // 肯定是平底三角形
    // 计算差值和梯度
    dy = (y1 - y0);
    dxdyl = ((x1 - x0)  << FIXP16_SHIFT)/dy;
    dudyl = ((tu1 - tu0) << FIXP16_SHIFT)/dy;
    dvdyl = ((tv1 - tv0) << FIXP16_SHIFT)/dy;
    dwdyl = ((tw1 - tw0) << FIXP16_SHIFT)/dy;
    dxdyr = ((x2 - x0)  << FIXP16_SHIFT)/dy;
    dudyr = ((tu2 - tu0) << FIXP16_SHIFT)/dy;
    dvdyr = ((tv2 - tv0) << FIXP16_SHIFT)/dy;
    dwdyr = ((tw2 - tw0) << FIXP16_SHIFT)/dy;

    // 垂直裁剪测试
    if (y0 < min_clip_y)
      {
      // 重新计算差值
      dy = (min_clip_y - y0);
      // 计算第一条扫描线起点的各种值
      xl = dxdyl*dy + (x0 << FIXP16_SHIFT);
      ul = dudyl*dy + (tu0 << FIXP16_SHIFT);
      vl = dvdyl*dy + (tv0 << FIXP16_SHIFT);
      wl = dwdyl*dy + (tw0 << FIXP16_SHIFT);
      // 计算第一条扫描线终点的各种值
      xr = dxdyr*dy + (x0 << FIXP16_SHIFT);
      ur = dudyr*dy + (tu0 << FIXP16_SHIFT);
      vr = dvdyr*dy + (tv0 << FIXP16_SHIFT);
      wr = dwdyr*dy + (tw0 << FIXP16_SHIFT);
      // 计算第一条扫描线的 y 坐标
      ystart = min_clip_y;
      } // end if
```

```
else
  {
  // 不用裁剪
  // 设置第一条扫描线起点和终点的各种值
  xl = (x0 << FIXP16_SHIFT);
  xr = (x0 << FIXP16_SHIFT);
  ul = (tu0 << FIXP16_SHIFT);
  vl = (tv0 << FIXP16_SHIFT);
  wl = (tw0 << FIXP16_SHIFT);
  ur = (tu0 << FIXP16_SHIFT);
  vr = (tv0 << FIXP16_SHIFT);
  wr = (tw0 << FIXP16_SHIFT);
  // 设置第一条扫描线的 y 坐标
  ystart = y0;
  } // end else
} // end else flat bottom
// 总是检测三角形最下面的部分是否会被裁剪掉
if ((yend = y2) > max_clip_y)
  yend = max_clip_y;

  // 水平裁剪测试
if ((x0 < min_clip_x) || (x0 > max_clip_x) ||
  (x1 < min_clip_x) || (x1 > max_clip_x) ||
  (x2 < min_clip_x) || (x2 > max_clip_x))
{
  // 裁剪版本
// 让指针 screen_ptr 指向第一条扫描线起点在缓存中的位置
screen_ptr = dest_buffer + (ystart * mem_pitch);
for (yi = ystart; yi<=yend; yi++)
  {
  // 计算扫描线起点和终点的 x 坐标
  xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
  xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
  // 计算扫描线起点和终点的 x 坐标
ui = ul + FIXP16_ROUND_UP;
vi = vl + FIXP16_ROUND_UP;
wi = wl + FIXP16_ROUND_UP;
// 计算扫描线上的 R、G、B 梯度
if ((dx = (xend - xstart))>0)
  {
  du = (ur - ul)/dx;
  dv = (vr - vl)/dx;
  dw = (wr - wl)/dx;
  } // end if
else
  {
  du = (ur - ul);
  dv = (vr - vl);
  dw = (wr - wl);
  } // end else

//////////////////////////////////////////////////////////////////////////////

// 扫描线起点水平裁剪测试
if (xstart < min_clip_x)
  {
  // 计算起点移动距离
  dx = min_clip_x - xstart;

  // 重新计算扫描线起点的 R、G、B 值
  ui+=dx*du;
  vi+=dx*dv;
  wi+=dx*dw;

  // 重新设置循环起始条件
  xstart = min_clip_x;
  } // end if
```

```
// 终点水平裁剪测试
if (xend > max_clip_x)
  xend = max_clip_x;

////////////////////////////////////////////////////////////////////////////

// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
{
// 绘制像素，假设格式为 5.6.5
    screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
                     ((vi >> (FIXP16_SHIFT+2)) << 5) +
                      (wi >> (FIXP16_SHIFT+3));

  // 计算下一个像素的 R、G、B 值
  ui+=du;
  vi+=dv;
  wi+=dw;
  } // end for xi

   // 计算下一条扫描线起点和终点的 x 坐标和 R、G、B 值
   xl+=dxdyl;
   ul+=dudyl;
   vl+=dvdyl;
   wl+=dwdyl;
   xr+=dxdyr;
   ur+=dudyr;
   vr+=dvdyr;
   wr+=dwdyr;
   // 让指针 screen_ptr 指向视频缓存的下一行
   screen_ptr+=mem_pitch;
   } // end for y
    } // end if clip

// 三角形没有被裁剪时的绘制代码，由于篇幅有限，将其删除

 } // end if

// 绘制常规三角形的代码，由于篇幅有限，将其删除

} // end Draw_Gouraud_Triangle16
```

上面列出的是 16 位版本的 Gouraud 着色函数——Draw_Gouraud_Triangle16()，它没有涉及光照方面，只是根据 lit_color[ ]中的顶点颜色，通过插值计算三角形中各个像素的颜色。一个不好的消息是，作者决定从现在开始，在速度至关重要时假定图形卡使用的是 5.6.5 格式。换句话说，在此之前，作者总是检测当前模式或使用函数指针来访问正确的位合并（bit-merging）函数，这会降低速度。由于 99.9%的图形卡都使用 5.6.5 格式，上述假定不会导致问题。然而，如果读者的图形卡使用的是 5.5.5 格式，可以重新编写代码，以手工方式执行位操作。例如，在前面的函数中，绘制扫描线的代码如下：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
  // 绘制像素，假设格式为 5.6.5
  screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
           ((vi >> (FIXP16_SHIFT+2)) << 5) +
           (wi >> (FIXP16_SHIFT+3));
  // 计算下一个像素的 R、G、B 值
  ui+=du;
  vi+=dv;
  wi+=dw;
  } // end for xi
```

这里以手工方式写入 5.6.5 像素。之所以这样做，是因为不愿意在这个内循环中调用任何函数。如果要

修改它，必须修改所有在屏幕上绘制像素的代码；这些代码很容易找到，因为它们使用了移位和屏蔽运算，同时作者对其进行了注释。

注意：读者可能想知道，这里为何对于颜色分量使用的变量名为 *u*、*v* 和 *w*。这是因为作者首先编写了纹理映射函数，然后在此基础上编写了 Gouraud Shader，且没有将前者使用的 *u*、*v* 和 w 重命名为 *r*、g 和 b。因此，*u*、*v*、*w* 实际上表示的是红、绿、蓝分量。它们只是变量名而已，如果这些变量名让您不舒服，可以修改它们。不过修改量很大，即使使用文本替换，对它们进行修改无疑是自找麻烦。

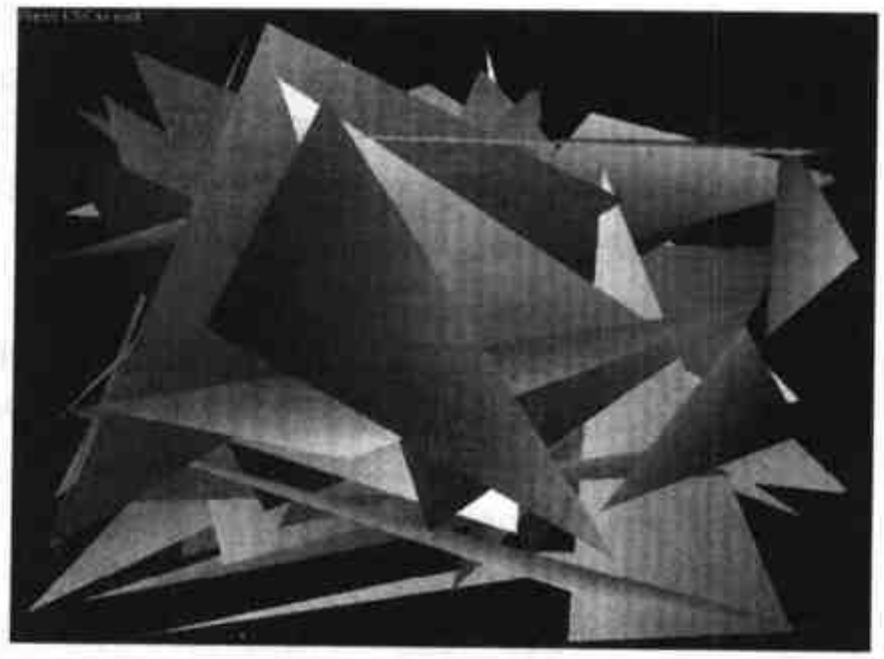


图 9.26　使用 Gouraud Shader 绘制三角形的演示程序

下面来看一个使用 Gouraud Shader 的演示程序，该演示程序只是绘制三角形而已。图 9.26 是 DEMOII9_1.CPP|:EXE 运行时的屏幕截图。读者可尝试运行该程序，它随机地选择三角形顶点的位置和颜色，然后调用 Gouraud Shader 绘制三角形。DEMOII9_1_8b.CPP|EXE 是一个使用 8 位版本的 Gouraud Shader 的演示程序。

9.5.2　对使用 Gouraud Shader 的多边形执行光照计算

接下来要做的是，使用光照引擎函数计算每个顶点的光照强度，得到每个顶点的最终 RGB 颜色。调用 Gouraud Shader 而不是标准的恒定 Shader 时，将得到随场景中的光照条件而变化的 Gouraud 着色三角形。下面回顾一下为此需要做哪些工作。我们已经创建了一种新的多边形结构，它包含一个颜色数组，用于存储经过光照处理后的顶点颜色。这个数组名为 lit_color[ ]，包含三个元素，每个三角形顶点一个。因此，我们只需修改对渲染列表或多边形进行光照处理的函数，使之知道如何对使用固定着色、恒定着色和 Gouraud 着色的多边形进行光照计算。

这些函数已经具备对使用固定和恒定着色的多边形进行光照计算的功能。对于使用固定着色的多边形，函数将多边形的颜色复制到 lit_color[0]中，仅此而已。对于使用恒定着色的多边形，函数使用系统中所有的活动光源，根据多边形的初始颜色和法线进行光照计算；因此，我们只需将这些代码复制到另一个循环中，使该循环对使用 Gouraud 着色的多边形的每个顶点进行光照计算。图 9.27 说明了多边形沿流水线进入光照阶段后，光照引擎对每个多边形执行光照计算的步骤。

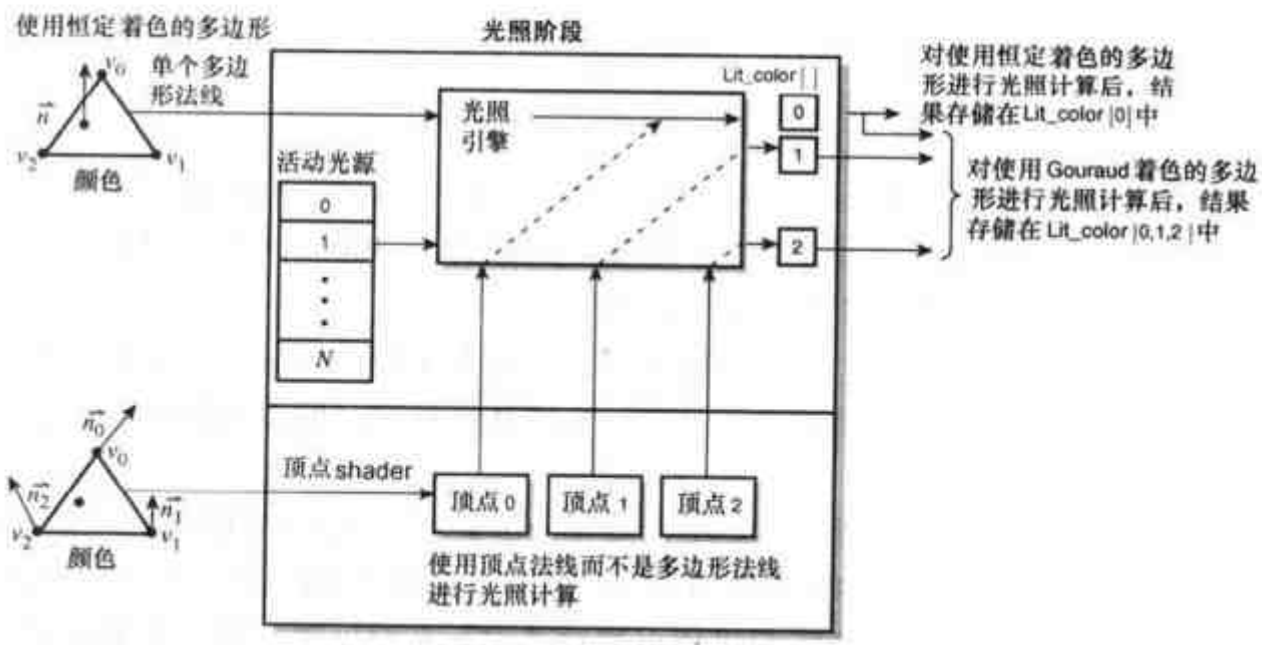


图 9.27　新的光照引擎支持恒定着色和 Gouraud 着色

从图中可知，为对使用 Gouraud 着色的多边形进行光照计算，只需复制对使用恒定着色的多边形进行光照计算的代码，并对其进行修改，使之使用存储在 VERTEX4DV1 结构中的 *n* 字段中的法线。当然，需要对多边形的每个顶点执行这样的计算，共 3 次。对于这些代码可以进行一些优化，这将在后面讨论。

下面来看看支持 Gouraud 着色的光照函数，这包括 8 位和 16 位模式的版本。

1. 16 位模式下的 Gouraud 光照计算

正如前面指出的，在光照函数中增加 Gouraud 光照计算支持很简单，只需复制执行恒定光照计算的代码，并对其进行修改，使之使用顶点法线而不是多边形法线，且循环执行这些代码 3 次（每个顶点一次）即可。然而，这将使光照函数变得非常复杂，同时对每个多边形进行光照处理所需的时间将很长。这也是基于软件的、支持 Gouraud 着色的彩色光照系统不多的原因。不过我们编写的光照函数还不错，即使是对于游戏而言，其速度也足够快。

为支持 Gouraud 光照，需要重新编写两个函数：基于物体的光照函数和基于渲染列表的光照函数。这里将介绍基于渲染列表的光照函数，因为我们的大部分工作是在渲染列表级完成的，包括光照计算。虽然在引擎的前一个版本中，提供了 8 位和 16 位模式的基于物体和渲染列表的光照函数，但我们通常使用基于渲染列表的光照函数。然而，对于 Gouraud 光照，必须从优化的角度重新审视这种做法。

请看图 9.28，其中的立方体被表示为物体和渲染列表中独立的多边形。问题在于，表示为 OBJECT4DV2 时，立方体只有 8 条顶点法线（因为只有 8 个顶点）；但立方体被分解为多边形并插入到渲染列表中后，将有 36（12*3）条顶点法线。这是由于渲染列表中的每个多边形都是自包含的，完全消除对外部顶点列表的引用。

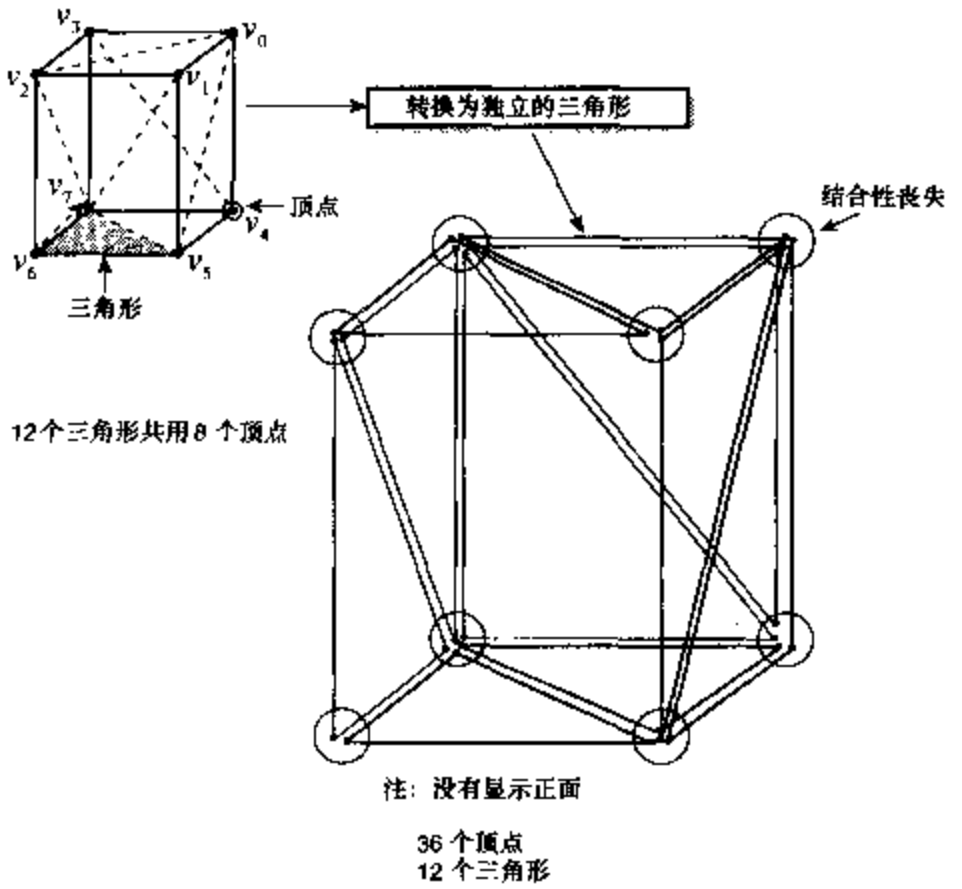


图 9.28　将物体转换为多边形后，顶点结合性将丧失

在 OBJECT4DV2 阶段（此时物体还是一个整体）执行光照计算时，顶点法线数最少；如果等到物体被转换为多边形后，对渲染列表执行光照计算，计算量将增加大约 300%，其中很多顶点法线是重复的。这显然是不能接受的，后面讨论优化时将解决这种问题。

在基于渲染列表的光照函数中，对于每个多边形都需要对 3 个顶点执行光照计算。虽然可以对执行了光照计算的顶点进行标记，以最大限度地减少计算量，这将非常复杂，所以现在暂时不去管它，但读者对此应该心中有数。

复杂的原因在于，对顶点执行光照处理，以便计算顶点的最终颜色并将其存储到 lit_colors[ ]时，需要使用包含该顶点的多边形的初始颜色。因此，要重用已经执行光照处理的顶点，必须将光照计算分成两步：第一步只计算光照强度，并对顶点进行标记，以免再次对其进行光照处理。第二步是使用多边形的初始颜色对顶点的光照强度进行调制。这将导致代码极其混乱，目前需要处理的问题已经很多，因此暂时不去管它。在后续章节中，将据此对代码进行优化。

现在请查看附带光盘中的 16 位光照函数。这些函数的代码太多（超过 30 页），这里无法列出它们。这两个新函数的原型如下：

```
int Light_RENDERLIST4DV2_World16(RENDERLIST4DV2_PTR rend_list,// 渲染列表
            CAM4DV1_PTR cam,   // 相机
            LIGHTV1_PTR lights, // 光源列表
            int max_lights)   // 最大光源数

int Light_OBJECT4DV2_World16(OBJECT4DV2_PTR obj,// 要处理的物体
            CAM4DV1_PTR cam,  // 相机
            LIGHTV1_PTR lights,// 光源列表
            int max_lights)  // 最大光源数
```

它们与前一个版本基本相同，只是针对新的物体结构和渲染列表结构进行了修改而已。下面是函数中使用环境光源和无穷远光源对采用 Gouraud 着色的多边形的每个顶点执行光照处理的代码：

```
if (curr_poly->attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD) ////////////////////
 {
 // 多边形使用 Gouraud 着色
 // 在流水线的这个阶段，网格已经被分解为三角形，
 // 因此如果顶点被多个三角形共享，将被执行光照计算多次
 // 使用 Gouraud 着色时，在物体级执行光照计算更合适
 // 因为此时顶点的共用性没有被破坏，光照计算与使用恒定着色类似
 // 只是需要执行 3 次，每个顶点一次
 // 另外，还有很多可优化的地方，但为确保代码易于理解
 // 这里暂时不管它，以后再进行优化

 // 第 1 步：提取多边形初始颜色的 R、G、B 值
 // 假设颜色格式为 5.6.5
_RGB565FROM16BIT(curr_poly->color, &r_base, &g_base, &b_base);

 // 扩展成 8 位
 r_base <<= 3;  g_base <<= 2;  b_base <<= 3;

 // 初始化顶点的累积 R、G、B 值
 r_sum0 = 0;  g_sum0 = 0;  b_sum0 = 0;
 r_sum1 = 0;  g_sum1 = 0;  b_sum1 = 0;
 r_sum2 = 0;  g_sum2 = 0;  b_sum2 = 0;

 // 优化措施：
 // 系统中有多个光源时，很多计算是重复的
 // 为最大限度地减少这种计算，作者采取的策略是，将需要重复计算的量设置为最大值
 // 然后在光照计算循环中检测这个变量是否为最大值
 // 如果是，则计算相应的量，并将其存储到该变量中
 // 这样，在以后的光源需要这个数据时，直接使用它即可
 for (int curr_light = 0; curr_light < max_lights; curr_light++)
  {
  // 光源是否被处于活动状态
  if (lights[curr_light].state==LIGHTV1_STATE_OFF)
   continue;

  // 判断光源的类型
  if (lights[curr_light].attr & LIGHTV1_ATTR_AMBIENT) ////////////
   {
   // 将光源颜色的各个分量与多边形颜色的相应分量相乘
   // 然后除以 256，使结果的取值范围为 0-255，在代码中使用移位（>> 8）
   ri = ((lights[curr_light].c_ambient.r * r_base) / 256);
   gi = ((lights[curr_light].c_ambient.g * g_base) / 256);
   bi = ((lights[curr_light].c_ambient.b * b_base) / 256);

   // 环境光源对每个顶点的影响相同
   r_sum0+=ri; g_sum0+=gi; b_sum0+=bi;
   r_sum1+=ri; g_sum1+=gi; b_sum1+=bi;
   r_sum2+=ri; g_sum2+=gi; b_sum2+=bi;
   } // end if
  else
  if (lights[curr_light].attr & LIGHTV1_ATTR_INFINITE) //////////
   {
   // 无穷远光源，需要使用顶点法线和光源的方向
   // 顶点法线已经计算好，其长度为 1.0，因此无需重新计算
```

```
// 无穷远光源的光照模型如下：
// I(d)dir = IOdir * Cldir
// 对于散射光，计算公式如下：
// Itotald =  Rsdiffuse*Idiffuse * (n . l)
// 为避免浮点数运算，将 dp 乘以 128
// 这样做并非浮点数运算速度更慢，而是因为数据类型转换会占用大量的 CPU 周期

// 需要对每个顶点执行光照计算

// 顶点 0
dp = VECTOR4D_Dot(&curr_poly->tvlist[0].n,
          &lights[curr_light].dir);

// 仅当 dp > 0 时，光源对顶点才有影响
if (dp > 0)
  {
  i = 128*dp;
  r_sum0+= (lights[curr_light].c_diffuse.r * r_base * i) /
      (256*128);
  g_sum0+= (lights[curr_light].c_diffuse.g * g_base * i) /
      (256*128);
  b_sum0+= (lights[curr_light].c_diffuse.b * b_base * i) /
      (256*128);
  } // end if

// 顶点 1
dp = VECTOR4D_Dot(&curr_poly->tvlist[1].n,
          &lights[curr_light].dir);

// 仅当 dp > 0 时光源对顶点才有影响
if (dp > 0)
  {
  i = 128*dp;
  r_sum1+= (lights[curr_light].c_diffuse.r * r_base * i) /
      (256*128);
  g_sum1+= (lights[curr_light].c_diffuse.g * g_base * i) /
      (256*128);
  b_sum1+= (lights[curr_light].c_diffuse.b * b_base * i) /
      (256*128);
  } // end if

// 顶点 2
dp = VECTOR4D_Dot(&curr_poly->tvlist[2].n,
          &lights[curr_light].dir);

// 仅当 dp > 0 时光源对顶点才有影响
if (dp > 0)
  {
  i = 128*dp;
  r_sum2+= (lights[curr_light].c_diffuse.r * r_base * i) /
      (256*128);
  g_sum2+= (lights[curr_light].c_diffuse.g * g_base * i) /
      (256*128);
  b_sum2+= (lights[curr_light].c_diffuse.b * b_base * i) /
      (256*128);
  } // end if

} // end if infinite light
```

在上述代码中，r_sum0 到 r_sum2、g_sum0 到 g_sum2 以及 b_sum0 到 b_sum_2，分别用于存储经过光照处理后，顶点 0 到 2 的颜色的红、绿、蓝分量。r_base、g_base、b_base 分别是多边形的初始颜色的 R、G、B 分量。计算方法与使用恒定着色时相同，只是需要重复 3 次。从无穷远光源模型的光照计算代码可知，对于每个顶点的光照计算是相同的。

在新的光照函数中，进行了非常有趣的优化，作者称之为保存计算结果（calculation caching）。通过分析对多边形进行光照处理的循环可以发现，这里涉及到多个光源：环境光源、无穷远光源、点光源等。对

于每个光源，以前都重新计算所需的数据；也就是说，需要知道光源到多边形的距离时，重新计算它，需要法线时，重新计算它；等等。这里进行的优化是，存储那些不随光源而异的数据，如多边形法线。这样做可以提高光照计算的速度，但只适用于有多个光源的情况。这里没有存储过多的数据，而只是存储了点光源和聚光灯光照计算中都需要的多边形法线，后面讨论优化时，将采取更为积极的态度。

不幸的是，在执行 Gouraud 着色光照计算的代码中，不能采用这种优化方法；但在执行恒定着色光照计算的代码中可以这样做。下面列出光照函数中采用这种优化方法执行恒定着色光照计算的代码，首先是进入光源循环之前的代码：

```
// 将 n.z 设置为 FLT_MAX，以指出法线未被计算
n.z = FLT_MAX;

// 遍历光源
for (int curr_light = 0; curr_light < max_lights; curr_light++)
  {
```

上述代码将法线向量的一个分量设置为无穷大，指出法线还未计算过。在光照计算循环中，需要使用法线的光照模型包含下述代码：

```
// 检查之前是否计算了多边形法线
if (n.z==FLT_MAX)
  {
  // 需要计算多边形的面法线
  // 顶点按顺时针方向排列，因此 u=p0->p1，v=p0->p2，  n=uxv

  // 计算 u 和 v
  VECTOR4D_Build(&curr_poly->tvlist[0].v, &curr_poly->tvlist[1].v, &u);
  VECTOR4D_Build(&curr_poly->tvlist[0].v, &curr_poly->tvlist[2].v, &v);

  // 计算叉积
  VECTOR4D_Cross(&u, &v, &n);
  } // end if
```

上述代码检查 n.z 是否为 FLT_MAX，如果是，则计算法线；否则不计算。这样，即使系统中有 8 个光源，也只计算法线一次。

提示：采用这种策略确实可以提高速度，前提是系统中有很多光源。然而，游戏中可能只有一两个光源，在这种情况下，这种优化实际上会降低速度。因此，如果要使用这种优化，必须确保其减少的计算量超过了带来的开销。

有关根据光照引擎计算得到的颜色值进行 Gouraud 着色的演示程序，请参阅 DEMOII9_2.CPP|EXE。图 9.29 是这个程序运行时的屏幕截图。这个演示程序创建 3 个水分子模型，然后在包含多个光源的场景中旋

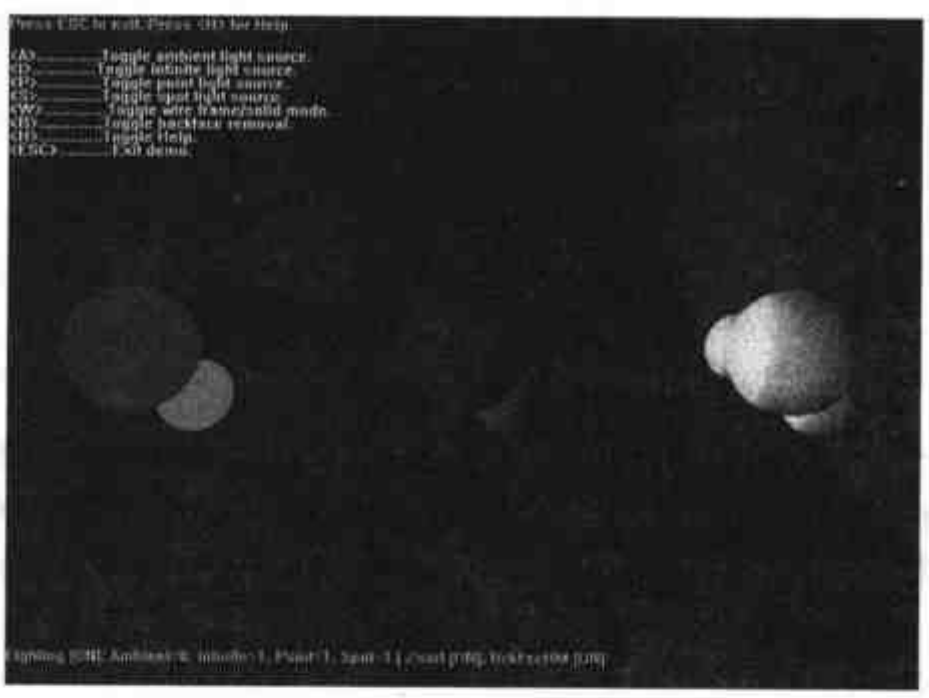


图 9.29　使用 Gouraud 光照计算的 3D 演示程序

转它们。3 个水分子模型分别使用固定着色、恒定着色和 Gouraud 着色——您能分别出各个模型使用的是哪种着色方法吗？使用 Gouraud 着色的分子模型看起来较圆润，虽然模型是由多个小平面组成的，这是通过平均计算顶点法线的结果。

2. 8 位模式下的 Gouraud 光照计算

在 8 位模式下，问题稍微复杂些。令人不可思议的是，8 位模式下的光照计算几乎与 16 位模式下相同。事实上，8 位版本的光照函数的代码几乎与 16 位版本相同，惟一的差别是，如果多边形使用的是固定或恒定着色，将使用查找表将最终的 RGB 颜色转换为颜色索引，并将其存储在 lit_color[0]中：

```
curr_poly->lit_color[0] = rgblookup[RGB16Bit565(r_sum, g_sum, b_sum)];
```

查找表的工作原理是，将 5.6.5 格式的 RGB 值作为一个 16 位的索引，根据查找表得到与其最接近的颜色索引。在程序的初始化阶段，调用下述函数来创建查找表：

```
// 创建供光照引擎使用的查找表
RGB_16_8_IndexedRGB_Table_Builder(DD_PIXEL_FORMAT565, // 像素格式
                  palette,  // 调色板
                  rgblookup); // 存储查找表的内存空间
```

这很好，但存在一个问题：对使用 Gouraud 着色的三角形执行光照计算时，将顶点的最终颜色转换为颜色索引是没有意义的，因为接下来进行 Gouraud 插值时，需要使用 RGB 颜色。因此，在 8 位模式下对使用 Gouraud 着色的三角形执行光照计算时，将把顶点的 RGB 颜色存储在数组元素 lit_color[ ]中，就像在 16 位模式下一样。接下来介绍 8 位版本的 Gouraud 着色函数。

在 8 位模式下，绘制使用 Gouraud 着色的三角形的函数如下：

```
void Draw_Gouraud_Triangle(POLYF4DV2_PTR face,// 指向多边形面的指针
        UCHAR *dest_buffer, // 指向视频缓存的指针
        int mem_pitch)   // 每行占据多少个字节（320、640 等）
```

它与 16 位版本基本相同，但将像素绘制到屏幕上的代码不同。这个函数的代码太多，无法全部列出它们，这里只介绍一些重要代码。首先来看 RGB 颜色提取步骤：

```
// 假设像素格式为 5.6.5
// 无法承受在循环中调用函数的开销
// 如果要支持 5.6.5 和 5.5.5 格式，需要编写两个硬编码版本
// 虽然将在 8 位模式下进行光栅化，但输入的颜色仍为 RGB 格式
_RGB565FROM16BIT(face->lit_color[v0], &r_base0, &g_base0, &b_base0);
_RGB565FROM16BIT(face->lit_color[v1], &r_base1, &g_base1, &b_base1);
_RGB565FROM16BIT(face->lit_color[v2], &r_base2, &g_base2, &b_base2);

// 扩展到 8 位
r_base0 <<= 3;
g_base0 <<= 2;
b_base0 <<= 3;

// 扩展到 8 位
r_base1 <<= 3;
g_base1 <<= 2;
b_base1 <<= 3;

// 扩展到 8 位
r_base2 <<= 3;
g_base2 <<= 2;
b_base2 <<= 3;
```

这与 16 位版本中完全相同，这是因为在 8 位光照函数中，对使用 Gouraud 着色的三角形进行光照计算时，将顶点的最终颜色以 RGB 5.6.5 格式存储在数组 lit_colors[ ]中，而没有将它们转换为索引。将 RGB 颜色转换为颜色索引只会浪费时间和 CPU 周期，因为无法对颜色索引进行插值。下面是光栅化扫描线的内循环：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
```

```
// write textel assume 5.6.5
screen_ptr[xi] = rgblookup[( ((ui >> (FIXP16_SHIFT+3)) << 11) +
                 ((vi >> (FIXP16_SHIFT+2)) << 5) +
                 (wi >> (FIXP16_SHIFT+3)) ) ];
// 设置下一个像素的 R、G、B 值
ui+=du;
vi+=dv;
wi+=dw;
} // end for xi
```

上述代码将 RGB 值转换为 5.6.5 格式，然后将其作为索引，在查找表中找到与之最接近的颜色索引。

这提出了一个哲学式问题。在 8 位模式下，速度要比 16 位模式下慢吗？如果采用简单的彩色光照，则答案是肯定的；如果使用专门为 8 位模式设计的单色光照，则答案是否定的。如果计算机的速度很慢，则不能使用 16 位模式和彩色光照，而只能支持白色光源，同时 Gouraud Shader 函数只需通过插值计算光照强度。在这种情况下，需要创建一个查找表，其中包含 256 种颜色，每种颜色有 256 种着色度，并使用颜色和强度作为这个查找表的索引。具有讽刺意味的是，本书前面创建过一个这样的查找表。因此，如果愿意，可以降低 8 位光照函数的功能，使之只计算每个顶点的光照强度。这样，8 位光栅化函数只需通过插值计算强度，与插值计算 RGB 相比，速度将快 3 倍。在光栅化阶段，根据颜色索引和强度在查找表中找到与之最接近的颜色索引。

还可以进行创新，创建多个查找表以支持彩色光源，如图 9.30 所示。然后，选择与场景中实际光源最接近的光源颜色，并通过相应的查找表来查找光照处理后的颜色。例如，假设只使用白色、红色、绿色、蓝色和橙色光源，则可以创建 5 个查找表，其中包含将各种强度的光照映射到不同颜色的表面时得到颜色索引。因此，可以使用查找表来实现彩色光照，而不影响性能。

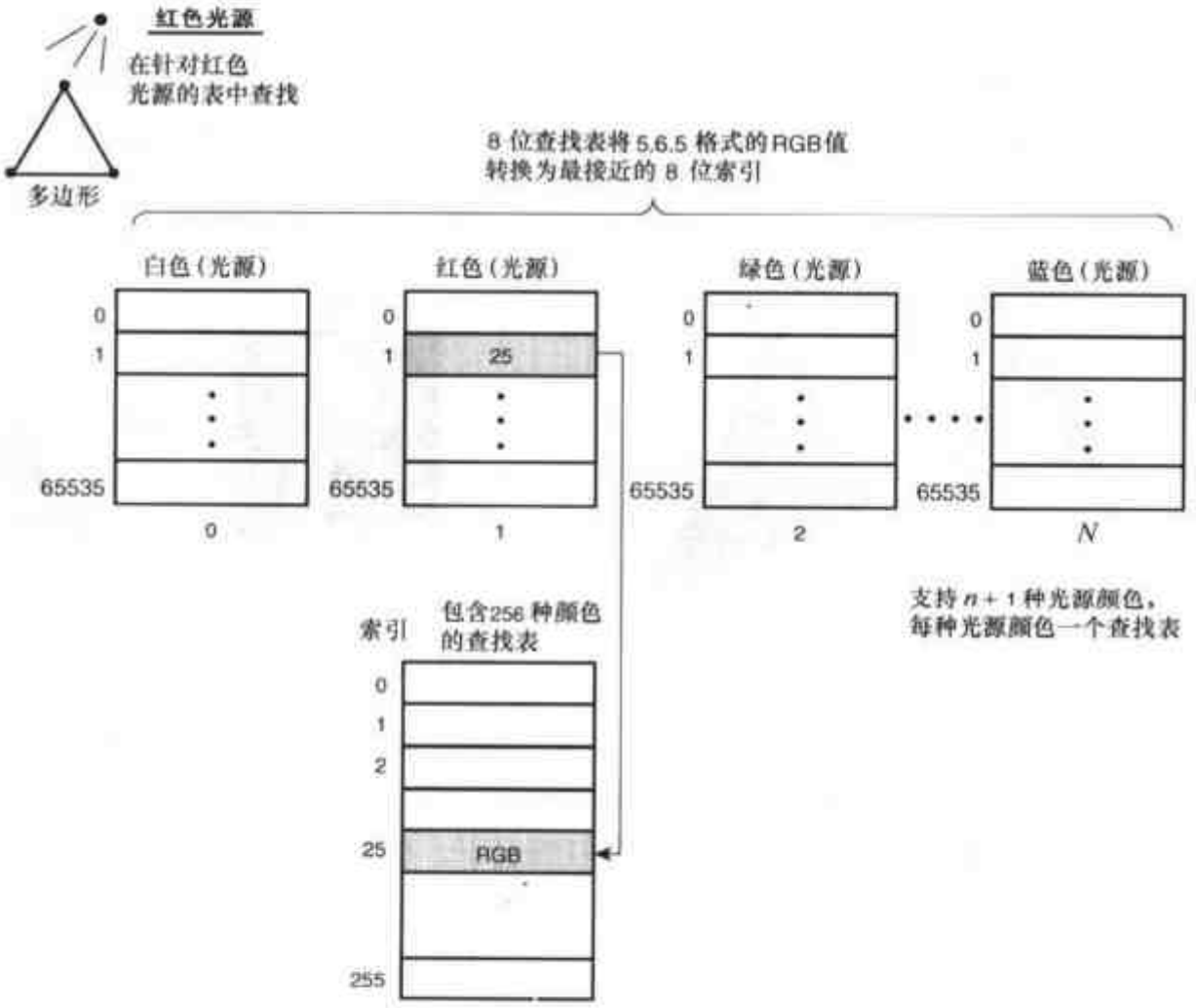


图 9.30　使用多个查找表，其中包含使用常见颜色的光源进行照射后多边形的颜色

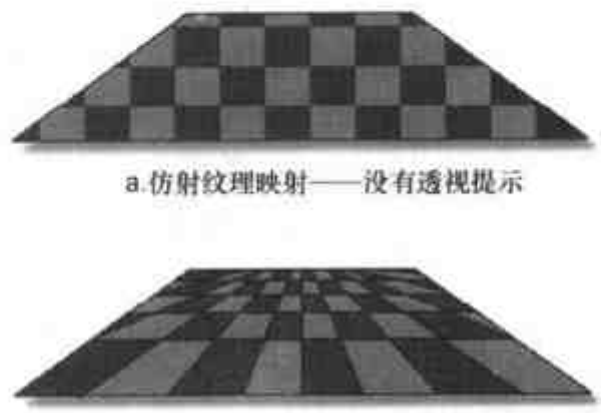
然而，完全没有必要在 8 位模式下这样做，因为当前 16 位模式下的速度已经足够快。事实上，使用 8 位模式实际上降低了速度，因为我们首先针对 16 位模式进行编程，然后将其转换为 8 位模式。另一方面，什么时候都可以使用查找表，没有人规定在 16 位模式下不能使用它。例如，如果只使用红色、绿色、蓝色、白色和橙色光源，便可以在 16 位的 Gouraud Shader 中使用单色插值，在函数的入口根据光源的颜色选择正确的查找表，从而避免进行 3 次插值。本书后面进行优化时将使用查找表，现在暂时不去管它。

有关使用 8 位 Gouraud 着色的演示程序，请参阅 DEMOII9_2_8b.CPP|EXE。请运行这个演示程序，它创建 3 个水分子模型，然后在包含多个光源的场景中旋转它们。3 个水分子模型分别使用固定着色、恒定着色和 Gouraud 着色。使用 Gouraud 着色的分子模型看起来更圆润，虽然 3 个模型是一样的。令人惊讶的是，虽然颜色空间有限，但效果还是不错的。

9.6 基本采样理论

您实际上已经完成了纹理映射工作！说正经的，纹理映射与 Gouraud 着色没什么差别，只是通过插值计算纹理坐标而不是颜色。当然，还涉及到一些细节，仅此而已。现在只讨论仿射（线性）纹理映射，完全不涉及透视修正。图 9.31 是一个经典的线性纹理映射和透视修正纹理映射范例，其中使用的纹理是一个棋盘。

然而，大可不必对纹理映射感到恐惧。读者已经知道如何实现纹理映射了。下面从另一个角度来审视插值——通过插值来计算坐标，进而根据坐标采集样本。



a.仿射纹理映射——没有透视提示

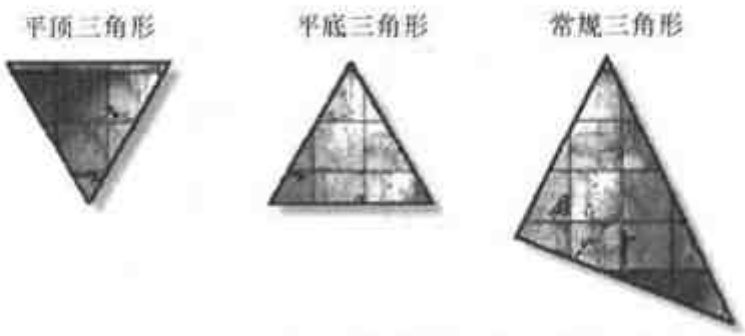
b.透视纹理映射——请注意近处和远处的 3D 透视

图 9.31 线性纹理映射和透视纹理映射

9.6.1 一维空间中的采样

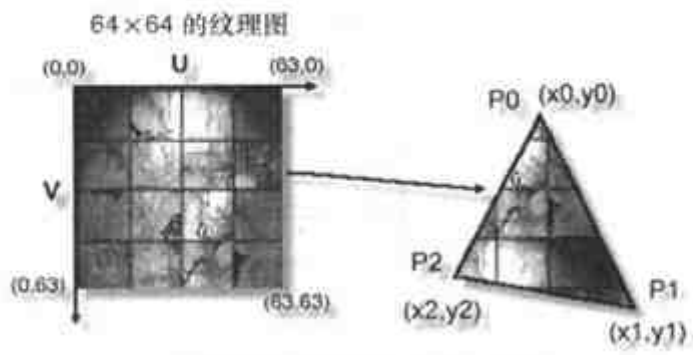
我们要做的是，设计一种对纹理图进行采样的算法，以便对三角形进行渲染时，可以使用采集到的像素对每条扫描线上的每个像素进行着色（color）。图 9.32 针对 3 类三角形（平顶三角形、平底三角形和常规三角形）说明了这一点。

图 9.33 说明了我们要实现的处理过程。我们将一个 m×m（8/16 位颜色）的矩形位图作为纹理映射到任何一个三角形上，这意味着需要考虑三角形的旋转和缩放。为帮助设计这种算法，图 9.33 中对一些感兴趣的点进行了标记。



注：根据那条边更长，常规三角形又分为两类

图 9.32 对 3 类基本三角形进行纹理映射



要将纹理映射到其上面的三角形

图 9.33 将 u、v 空间中的纹理映射到三角形上

首先，目标三角形由顶点 p0、p1 和 p2 构成，它们的坐标分别是(*x*0，*y*0)、(*x*1，*y*1)和(*x*2，*y*2)。另外，在纹理图周围标出了水平轴 u 和垂直轴 v，纹理左上角的坐标为(0，0)，右下角的坐标为(63，63)，因为坐标 *u* 和 *v* 的取值范围都为 0～63。然而，在很多情况下，坐标 u 和 *v* 的取值范围为 0～1（如果被归一化）或 0 到 2 的幂减 1。

将矩形纹理图以线性方式映射到三角形上只不过是进行插值而已，但插值次数很多，很容易出错或编写出速度很慢的算法。另外，虽然在讨论三角形光栅化和 Gouraud 着色时介绍过插值，但从未从采集数据的角度介绍过它。下面从另一个角度来讨论插值，即旨在采集数据而不是通过插值计算坐标或颜色。从最简单的情况——一维空间中的采样——开始。

图 9.34 是最简单的纹理映射方式：将纹理映射到一条垂直线上。在这个图中，有一个宽 1 像素、高 8 像素的纹理图，我们需要将它映射到一条宽 1 像素、高为任何值的多边形上。该如何办呢？可以求助于采样。

我们需要做的是，对纹理图（这里是一个 1×8 像素的位图）进行采样，并将其映射到一个 1×*n* 像素的目标多边形上，其中 *n* 的取值范围为 1 到无穷大。请看图 9.34 中的一些例子。

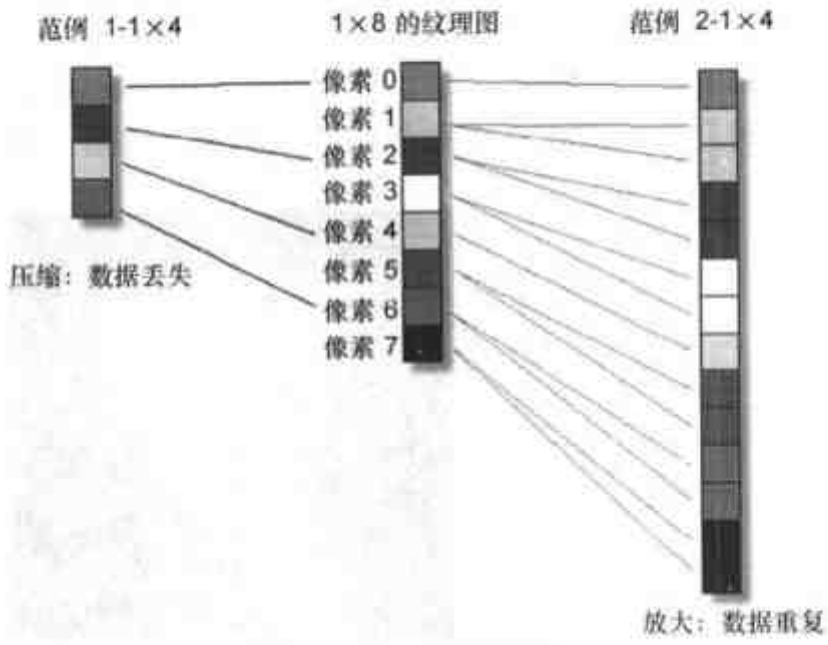


图 9.34　一维纹理映射

在范例 1 中，目标多边形为 1×4 像素的。在这种情况下，可以在纹理图的每两个像素中采集一个，如图中所示。假设选择像素 0、2、4 和 6，并将它们分别映射到目标多边形的 0、1、2 和 3 处，这样便完成了纹理映射。然而，选择像素 0、2、4 和 6 的决定是如何做出的呢？答案是根据采样比例，它是一个插值因子，计算公式如下：

采样比例 = 纹理图高度 / 目标多边形的高度

因此，在这个例子中，采样比例为：

采样比例 = 8 / 4 = 2.0

也就是说，沿目标多边形的垂直轴每移动 1.0 个像素，必须在纹理图中移动 2.0 个像素。这就是采样比例 2 的含义，因此采样序列为 0、2、4、6。读者可能会说，“等一等，我们丢失了颜色信息”。确实如此，必须丢弃一半的像素。这种采集整数阵列上的像素而不进行平均（滤波）的方法绝对存在问题。如果要编写像 3D Studio Max 这样的高端 3D 建模程序，可能需要对采集的像素进行平均（区域采样），使样本更接近于原始数据。然而，对于游戏和实时软件，第一种技术（点采样）是可行的。下面来看一个走向另一个极端的例子。

在范例 1 中，对纹理图进行了压缩，即目标多边形比纹理图小，因此将丢失信息。另一种情形是，目标多边形比纹理图大，没有足够的信息供多边形使用。在这种情况下，必须重复使用纹理图中的数据。在 3D 游戏中，当带纹理的多边形离视点太近时，看起来像是由很多块组成的。这是因为没有足够的纹理数据，有些采样点被采集多次，从而构成很多小块。请看图 9.34 中的第二个范例，纹理图仍然是 1×8 的，但目标多边形是 1×14 的。显然，在这种情况下，采样比例将小于 1：

采样比例 = 8 /14 = 0.57

因此，在目标多边形中绘制每个像素时，当前采样点与前一个点的距离应为 0.57 个像素。这样，对于目标像素 0～13，对应的采样点序列如下：

- 样本 0：0.57；
- 样本 1：1.14；
- 样本 2：1.71；
- 样本 3：2.28；
- 样本 4：2.85；
- 样本 5：3.42；
- 样本 6：3.99；
- 样本 7：4.56；
- 样本 8：5.13；
- 样本 9：5.7；
- 样本 10：6.27；
- 样本 11：6.84；
- 样本 12：7.41；
- 样本 13：7.98。

要得到实际的采样点，可以对计算得到的采样点进行截尾，结果为(0，1，1，2，2，3，3，4，5，5，6，6，7，7)。每个像素被采样大约两次（1/0.57）。

9.6.2　双线性插值

这里顺便简要地讨论一下基本滤波以及它为何重要。在前一个例子中，确定最终的样本像素时，将浮点数采样点进行截尾得到整型采样点。然而，如果考虑采样位置的小数部分，根据一个几何滤波器对多个

像素进行采样，情况将如何呢？图 9.35 说明了这一点，它描述了一个采用点采样得到的纹理和一个采用滤波得到的纹理。例如，对于前一个例子中的样本 12，滤波采样使用浮点数采样点 7.41 进行采样；但采用点采样时，将对采样点进行截尾，从而采集纹素（textel pixel）7.0，这将丢失信息。



A. 双线性插进　　B. 点采样

图 9.35　点采样和双线性滤波

点采样的速度更快，但效果不好。如果使用如图 9.36 所示的简单 1D 箱形滤波器，并假定像素中心的坐标为整数，则采样点 7.41 将导致箱形滤波器从 7.0 向前移动 0.41 个像素，使滤波器覆盖了像素 7.0 的一部分和像素 8.0 的一部分，覆盖率分别是 59%和 41%。这样，将使用一个混合函数来计算样本，如下所示：

$$样本 = 0.59 * 像素7_{rgb} + 0.41 * 像素8_{rgb}$$

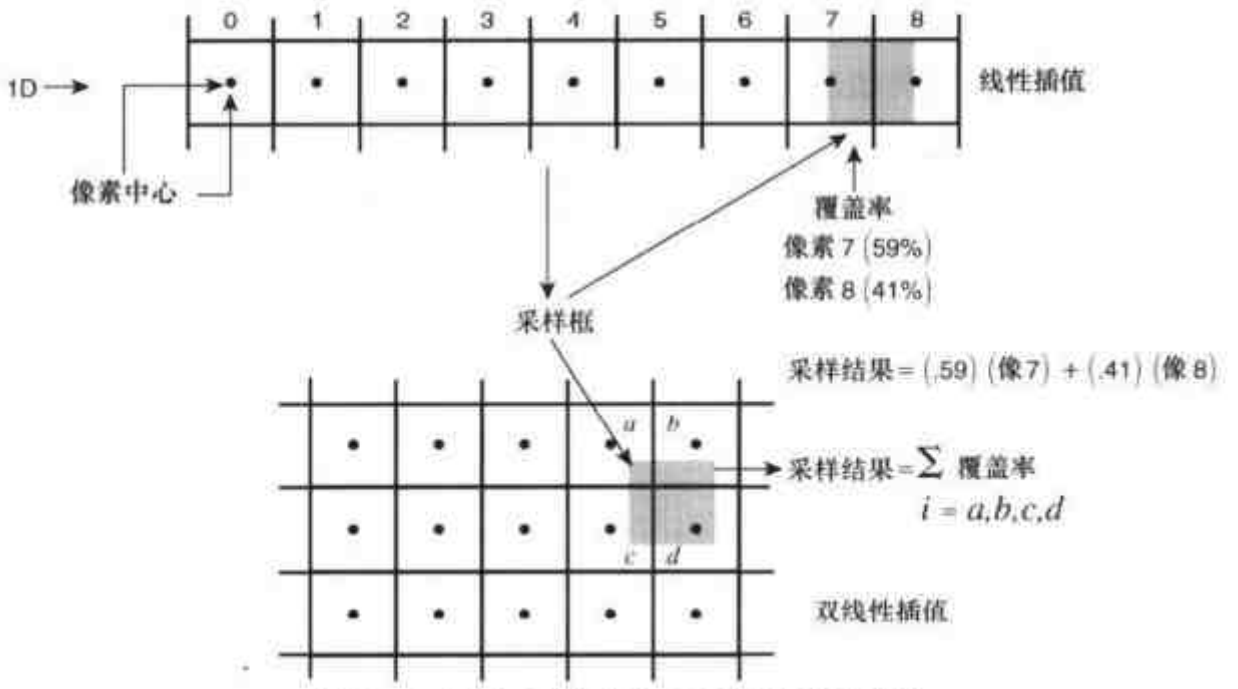


图 9.36　用于双线性插值的简单箱形滤波器

上述混合是在 RGB 空间中进行的，最后的样本为像素 7 和 8 的混合颜色。这样，效果将好得多，不会出现锯齿。这种采样方法的缺点是，每次采样都增加了一次插值，即混合运算，而不是进行简单的采样。虽然这没什么大不了的，但确实会降低速度。

另外，在实际的纹理映射函数中，是在二维空间而不是一维空间内执行这种操作，因此被称为双线性插值。然而，可以使用查找表和其他近似方法来提高其速度。例如，使用定点数进行插值计算时，可以将小数部分的 2～3 位用作混合因子，并创建一个包含每种颜色的各种比例的查找表，然后使用这个表来查找每个混合部分的值，并将它们相加。

提示：现在不打算实现双线性滤波，但以后可能实现它——如果能使其速度足够快的话。坦率地说，双线性滤波通常会使纹理不清晰，避免走样。然而，可以在引擎中预先对纹理进行柔化（softening），以最大限度地避免使用点采样得到的纹理走样。这样将丢失一些细节，但纹理看起来更美观，因此值得这样做。

9.6.3　u 和 v 的插值

我们要实现的算法是，在三角形的左侧边和右侧边上通过插值计算每条扫描线的起点和终点，然后根据合适的纹素绘制扫描线上的每个像素。首先需要给目标三角形的顶点指定纹理坐标，为插值提供参照系。因此，必须给每个顶点制定纹理坐标(*u*，*v*)，如图 9.37 所示。这样，每个顶点都包含 4 个坐标分量 *x*、*y*、*u* 和 *v*，将根据它们来进行纹理映射。新结构 VERTEX4DV1 支持纹理坐标，而结构 OBJECT4DV2 中也包含存储纹理坐标 u 和 v 的数组，因此定义纹理映射的准备工作已经就绪。

下面讨论一下纹理坐标的范围。纹理图的大小必须是 *m*×*m*（其中 *m* 为 2 的幂），通常为 64×64 像素、128×128 像素或 256×256 像素，这意味着纹理坐标为 0 到 *m*–1。为方便讨论，假设所有纹理图都是 64×64 像素的，因此任何顶点的纹理坐标必须是(0...63，0...63)。

例如，在图 9.38 中有两个纹理映射的例子。在第一个例子中，三角形的顶点 0、1 和 2 的纹理坐标分别是(0，0)、(63，0)和(63，63)，这样纹理图的一半将被映射到目标三角形上。在第二个例子中，纹理图被映射到两个相邻的三角形上，这两个三角形构成了一个正方形。纹理坐标被设置成这样：将纹理图的一半映射到其中一个三角形上，将另一半映射到另一个三角形上，这样，便将纹理完美地映射到了两个三角形上。另外，这也是使用三角形构建四边形的方法。

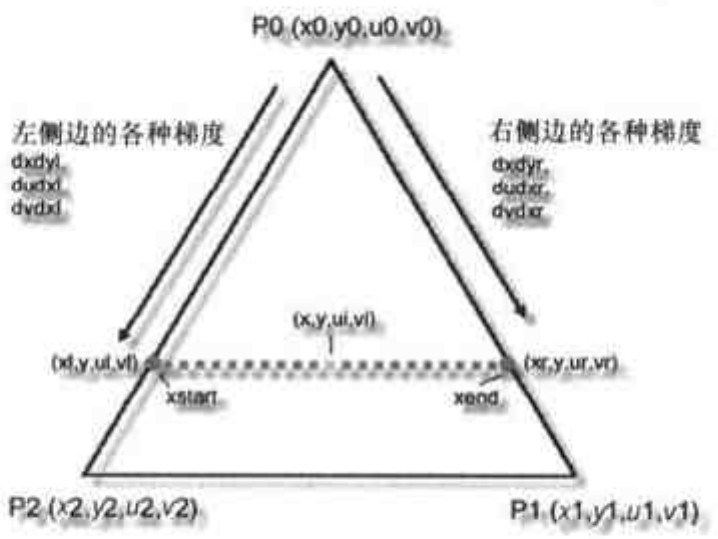


图 9.37　纹理映射

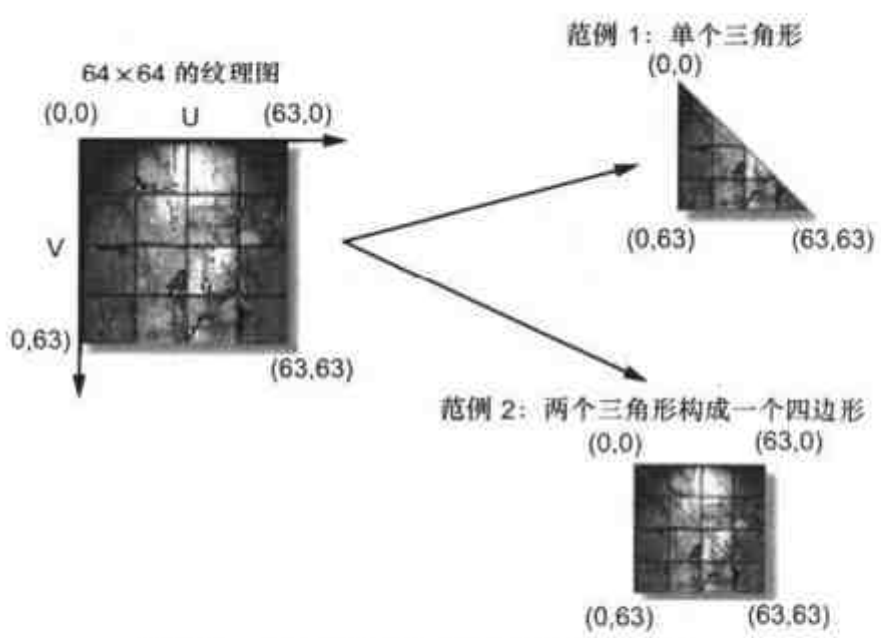


图 9.38　将纹理映射到三角形和四边形上

读者对问题有了直观的认识后，下面来实现其算法。在接下来的分析中，使用的变量名与图 9.37 标出的相同；同时，为方便读者理解程序代码，最后的纹理映射函数中也将使用这些变量名。

在左侧边上，各种梯度如下：

```
dxdyl = (x2 - x0)/(y2 - y0); // 左侧边的 x 坐标梯度
dudyl = (u2 - u0)/(y2 - y0); // 左侧边的 u 纹理坐标梯度
dvdyl = (v2 - v0)/(y2 - y0); // 左侧边的 v 纹理坐标梯度
```

在右侧边上，各种梯度如下：

```
dxdyr = (x1 - x0)/(y2 - y0); // 右侧边的 x 坐标梯度
dudyr = (u1 - u0)/(y2 - y0); // 右侧边的 u 纹理坐标梯度
dvdyr = (v1 - v0)/(y2 - y0); // 右侧边的 v 纹理坐标梯度
```

当然，可优化的地方很多。例如，（y2 - y0）项被使用多次，但只需计算一次即可。另外，更好的方法是，计算（*y*2 – *y*0）的倒数，然后使用乘法运算来计算各种梯度。根据梯度进行插值时，必须有初始值作为参照，我们从最上面的顶点（顶点 0）开始，因此这样设置初始值：

```
xl = x0; // 左侧边上 x 坐标的初始值
ul = u0; // 左侧边上纹理坐标 u 的初始值
vl = v0; // 左侧边上纹理坐标 v 的初始值

xr = x0; // 右侧边上 x 坐标的初始值
ur = u0; // 右侧边上纹理坐标 u 的初始值
```

```
vr = v0; // 右侧边上纹理坐标 v 的初始值
```

至此，准备工作几乎已经就绪。可以这样沿左侧边和右侧边进行插值：

```
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
```

但是，对于每条扫描线，还必须通过线性插值来计算每个像素的纹理坐标 *ui* 和 *vi*，然后将它们作为索引，从纹理图中得到相应的纹素。为此，只需计算扫描线起点和终点的纹理坐标 *u* 和 *v*，然后使用 *dx* 来计算扫描线上的 *u*、*v* 坐标梯度：

```
xstart = xl;     // 扫描线起点的 x 坐标
xend = xr;     // 扫描线终点的 x 坐标
dx = (xend - xstart); // 差值 dx
du = (ul - ur)/dx;
dv = (vl - vr)/dx;
```

有了 du 和 dv 后，便可以沿扫描线通过插值计算每个像素的纹理坐标，并据此获得纹理图中相应纹素的颜色，然后使用该颜色绘制像素。代码如下：

```
// 设置扫描线起点的纹理坐标
ui = ul;
vi = vl;
// 沿扫描线从左到右进行插值
for (x = xstart; x <= xend; x++)
   {
   // 读取纹素的颜色
   pixel = texture_map[ui][vi];
   // 在(x,y)处绘制像素
   Plot_Pixel(x,y,pixel);

   // 计算下一个像素的纹理坐标 u 和 v
   ui+=du;
   vi+=dv;
   } // end for x
```

仅此而已。当然，在外循环中，还需要沿左侧边和右侧边向下，通过插值计算出每条扫描线的 *xl*、*ul*、*vl*、*xr*、*ur* 和 *vr*。

编写纹理映射函数时，需要解决上述伪代码函数中没有解决的细节；但正如前面指出的，Gouraud 光栅化函数是在纹理映射函数的基础上修改而成的。也就是说，作者首先编写了纹理映射函数，然后将其转换为 Gouraud Shader，因此读者对这个函数已经很熟悉了。另外，纹理映射函数和 Gouraud Shacer 使用的是定点数，这样可避免在内循环中绘制像素时进行浮点数和整数之间的转换，因为作者对编译器不信任。

浮点数运算的速度和整数运算一样快，但有时候，使用编译器生成的代码进行类型转换时，所需的时间可达数学运算的 10～100 倍。

9.6.4 实现仿射纹理映射

至此，读者已经具备了编写纹理映射函数所需的理论知识，下面讨论这些函数的细节，并列出它们的原型和部分代码。作者编写了 8 位和 16 位版本的纹理映射函数，其中比较有趣的是对纹理图进行采样以及在屏幕上绘制像素的内循环。Gouraud 光栅化函数通过插值计算 RGB 值，然后使用它来绘制像素；而纹理映射函数通过插值计算纹理坐标，然后根据纹理坐标对纹理图进行采样，并渲染像素，当然这里没有使用双线性滤波。

1. 16 位模式下的纹理映射

16 位的纹理映射函数与 Gouraud Shader 几乎相同，只是绘制像素的内循环有些不同，稍后将介绍。现

在来看一下函数的原型及其调用方法：

```
void Draw_Textured_Triangle16(POLYF4DV2_PTR face, // 指向多边形面的指针
          UCHAR *_dest_buffer,// 指向视频缓存的指针
          int mem_pitch) // 每行占据的字节数（320、640 等）
```

少即是多——Stephen Wolfram 认为，使用几行代码就能够描述整个宇宙——在很多情况下，复杂性是从简单性派生而来的。要调用该函数，只需传递一个 POLYF4DV2 包含顶点和纹理图 POLYF4DV2、目标缓存和内存跨距（memory pitch）。然后，很快便可以在屏幕上看到一个被裁剪过的、带纹理的三角形。

为让读者深入了解如何调用纹理映射函数，下面提供了一个范例，这些代码摘自一个渲染函数：

```
if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
  {
  // 设置顶点
  face.tvlist[0].x = (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
  face.tvlist[0].y = (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
  face.tvlist[0].u0 = (int)rend_list->poly_ptrs[poly]->tvlist[0].u0;
  face.tvlist[0].v0 = (int)rend_list->poly_ptrs[poly]->tvlist[0].v0;

  face.tvlist[1].x = (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
  face.tvlist[1].y = (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
  face.tvlist[1].u0 = (int)rend_list->poly_ptrs[poly]->tvlist[1].u0;
  ace.tvlist[1].v0 = (int)rend_list->poly_ptrs[poly]->tvlist[1].v0;

  face.tvlist[2].x = (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
  face.tvlist[2].y = (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
  face.tvlist[2].u0 = (int)rend_list->poly_ptrs[poly]->tvlist[2].u0;
  face.tvlist[2].v0 = (int)rend_list->poly_ptrs[poly]->tvlist[2].v0;

  // 指定纹理
  face.texture = rend_list->poly_ptrs[poly]->texture;

// 使用固定着色?
if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
  {
  // 以固定着色绘制带纹理的三角形
  Draw_Textured_Triangle16(&face, video_buffer, lpitch);
  } // end if
else
  {
  // 以恒定着色绘制三角形
  face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];
  Draw_Textured_TriangleFS16(&face, video_buffer, lpitch);
  } // end else
```

请注意其中以粗体显示 POLYF4DV2 的代码，它们设置三角形。正如读者看到的，指定了多边形面的每个顶点的坐标 x、y、u 和 v，然后将该多边形面作为参数来调用函数 Draw_Textured_Triangle16()。读者可能注意到了，随后还调用了函数 Draw_Textured_TriangleFS16()，这是支持恒定着色的函数版本，稍后将介绍它。当前，纹理映射函数 Draw_Textured_Triangle16()只支持固定着色。也就是说，在模型中，必须将着色模式设置为“constant”，然后应用一个纹理。场景中的光源对纹理没有任何影响。稍后将进行改进，第一个版本的纹理映射函数只支持固定着色。接下来看一下函数 Draw_Textured_Triangle16()中的一些重要代码。

下面是纹理映射函数的内循环：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
  // 绘制像素
  screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) + (
             (vi >> FIXP16_SHIFT) << texture_shift2)];
  // 计算下一个像素的纹理坐标 u 和 v
  ui+=du;
  vi+=dv;
  } // end for xi
```

下面讨论一下以粗体显示的代码。首先，纹理坐标 *ui* 和 *vi* 为 16.16 的定点格式，因此需要执行移位运算，这相当于下述代码：

```
screen_ptr[xi] = textmap[ui + vi * word_pitch]
```

乘以 word_pitch 是通过左移 texture_shift2 实现的；由于纹理图是以先行后列的方式存储的，因此可以使用上述表达式像访问 2D 数组那样访问它。texture_shift2 是如何推导出来的呢？这很简单：它是纹理图宽度的以 2 为底的对数。如果纹理图的大小为 64×64（纹理图总是方形的），则 texture_shift2 为 6，因为 $2^6 = 64$（$\log_2{}^{64} = 6$）。

稍微有些麻烦的是，作者使用结构 BITMAP_IMAGE 来存储纹理图，虽然这个结构中存储了纹理的宽度和高度，但并没有存储它们的以 2 为底的对数。可以将这些值存储到多边形结构中，或者将它们存储到 BITMAP_IMAGE 结构的某个字段的前 4 位中，但这样做太隐蔽。相反，作者创建了一个 $\log_2$ 查找表：

```
// logbase2ofx[x] = (int)log2 x, [x:0-256]
UCHAR logbase2ofx[257] =
{
0,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3, 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5, 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6, 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6, 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7, 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7, 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7, 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7, 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7, 8,
};
```

这个表的用法很简单：以 0～256 的任何整数作为索引，便可以得到与其以 2 为底的对数最接近的整数。例如，logbase2ofx[128] = 7，logbase2ofx[129]也为 7，等等。在纹理映射函数的开头，计算纹理图宽度的以 2 为底的对数，并将结果存储到 texture_shift2 中，然后在纹素时使用它来执行移位运算，这样便可以支持从 1×1 到 256×256 的任何纹理。然而，作者不会使用小于 16×16 的纹理，因为在这种情况下，定点数运算可能不正常。

这又是一个从非常规角度进行思考，使用查找表来提高速度的例子。然而，我们应该创建一个新的 BITMAP 结构，预先计算出纹理宽度的以 2 为底的对数，并将其存储到这个结构中。对于每个多边形执行一次表查找操作没有什么大不了的，但是别忘了，积沙成塔，不断加上稻草也会将骆驼压死。因此，一定要小心，我们在逐渐降低性能。

2. 8 位模式下的纹理映射

在 8 位模式下，支持固定着色的纹理映射函数与 16 位版本相同，因为它只是执行采样运算，而没有执行颜色运算。因此，代码并没有太多变化，只是数据类型为 BYTE，而不是 SHORT 而已。除此之外，其他方面完全相同，因此这里不再赘述，这个函数的原型如下：

```
void Draw_Textured_Triangle(POLYF4DV2_PTR face, // 指向多边形面的指针
        UCHAR *_dest_buffer, // 指向视频缓存的指针
        int mem_pitch) // 每行占据的字节数（320、640 等）
```

9.7 更新光照/光栅化引擎以支持纹理

现在来更新光照引擎以支持纹理。从很大程度上说，这项工作已经完成！为支持纹理，几乎不用在光照函数中增加任何代码。不过，确实需要耍一个小花样，且听我细细道来。

对多边形执行光照计算时，我们将多边形的初始颜色作为材质，使用光源对其进行调制。也就是说，红色多边形只反射红色光，等等。然而，我们不能对纹理图的每个像素执行光照计算，而需要玩一个小花样。我们将这样做：对于将被贴上纹理的多边形，将其颜色重新设置为白色(255，255，255)，然后像通常那样对其进行光照计算。然而，到目前为止，作者编写的光栅化函数只支持使用恒定着色的、带纹理的多

边形，而不支持使用 Gouraud 着色的、带纹理的多边形，因此带纹理的多边形只能使用固定或恒定着色。回到前面的话题，对于带纹理的多边形，将把它的颜色重新设置为白色，然后像通常那样执行光照计算，并忽略多边形是带纹理的。

这样，多边形将反射照射到其上的所有光线，这正是我们希望的。在光栅化阶段，绘制带纹理的多边形时，从纹理图中采集合适的纹理，并用经过光照处理后的多边形颜色对其进行调制。接下来看看 16 位和 8 位实现。

1．在 16 位纹理映射函数中添加光照处理功能

在 16 位模式下，对于使用恒定着色且带纹理的多边形，光照处理和光栅化流程如图 9.39 所示。绘制这种多边形的函数的原型如下：

```
void Draw_Textured_TriangleFS16(POLYF4DV2_PTR face, // 指向多边形面的指针
        UCHAR *_dest_buffer, // 指向视频缓存的指针
        int mem_pitch)    // 每行占据的字节数（320、640 等）
```

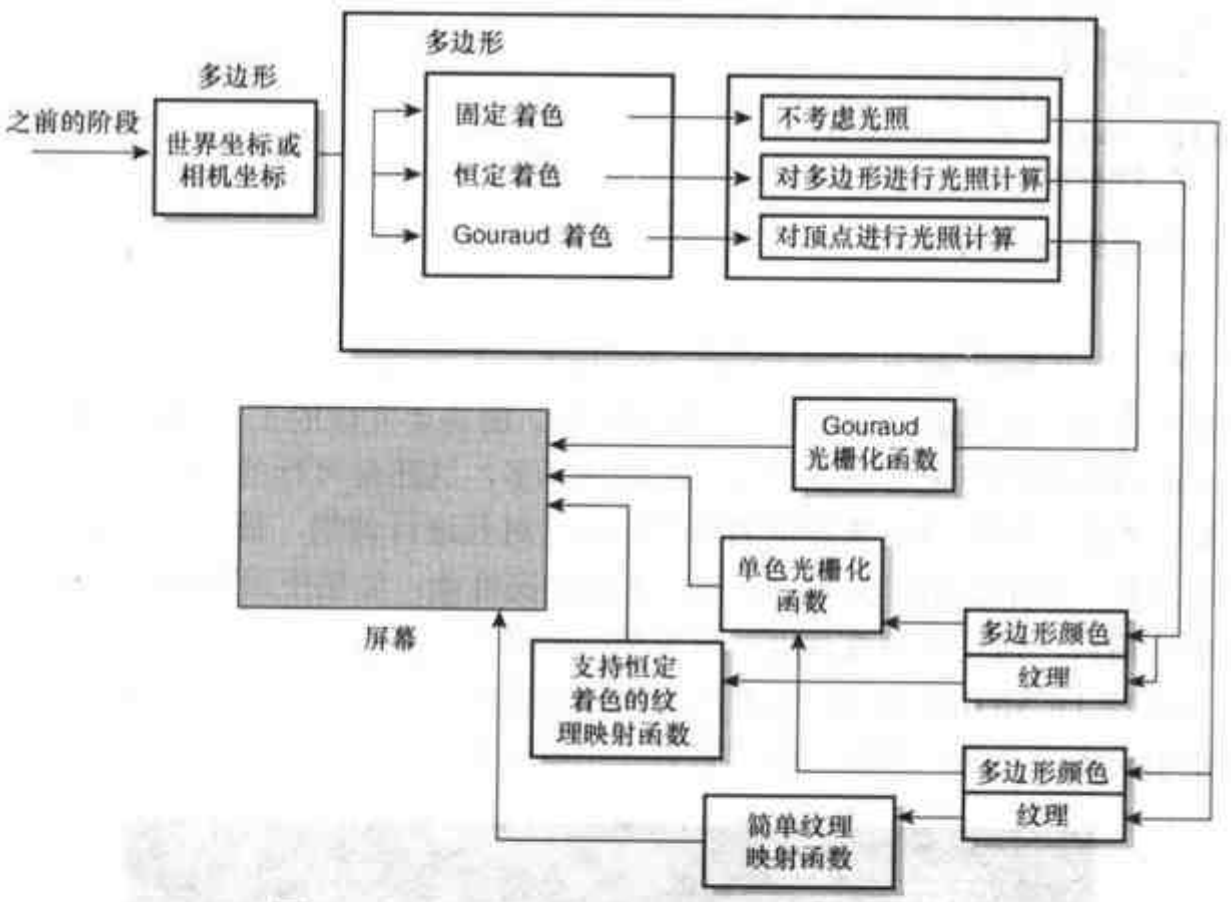


图 9.39　对纹理进行光照处理和光栅化的流水线

这个函数的原型与只支持固定着色的版本相同，只是函数名中包含“FS”，以指出它支持恒定着色。下面摘录其中的一些代码，让读者明白是如何对纹理进行着色的。前面讲过，光照函数假定多边形的颜色为白色，然后进行光照计算，并将结果存储在 lit_color[0]中。因此，我们只需对纹素颜色进行调制（将其与光照处理后的多边形颜色相乘），换句话说，对纹理进行光照处理。首先来看看支持固定着色的纹理映射函数的内循环代码：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
  // 绘制像素
  screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
                ((vi >> FIXP16_SHIFT) << texture_shift2)];
  // 计算下一个像素的纹理坐标 u 和 v
  ui+=du;
  vi+=dv;
  } // end for xi
```

下面是新的内循环代码，它使用光照处理后的颜色对纹素进行调制：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
```

```
// 绘制像素
// 首先读取纹素
textel = textmap[(ui >> FIXP16_SHIFT) +
       ((vi >> FIXP16_SHIFT) << texture_shift2)];

// 提取纹素颜色的 R、G、B 值
r_textel = ((textel >> 11)    );
g_textel = ((textel >> 5) & 0x3f);
b_textel =  (textel    & 0x1f);

// 使用经过光照处理后的多边形颜色进行调制
r_textel*=r_base;
g_textel*=g_base;
b_textel*=b_base;

// 最后绘制像素 hence we need to divide
// 前面将 R、G、B 分别放大了 32、64 和 32 倍，因此需要将它们分别除以 32、64 和 32
// 但由于需要将结果进行移位，以放到 5.6.5 字中
// 因此可以利用移位来完成
screen_ptr[xi] = ((b_textel >> 5) +
        ((g_textel >> 6) << 5) +
        ((r_textel >> 5) << 11));

// 计算下一个像素的纹理坐标 u 和 v
ui+=du;
vi+=dv;
} // end for xi
```

上述内循环将极大地降低性能。为支持恒定纹理映射，所需要完成的工作量比以前增加了 10 倍，但令人惊讶的是，只要游戏中使用恒定着色且带纹理的物体不多，这还是可行的。内循环完成的工作如下：从纹理图中提取纹素，然后使用光照引擎提供的多边形颜色对其进行调制，最后在屏幕上绘制像素。其中最费时的操作是移位运算。如果使用汇编语言，可极大地提高性能；如果使用 SIMD（单指令多数据）指令，可将速度提高 4 倍。最后，通过使用查找表，可避免一些数学运算。

图 9.40 是演示程序 DEMOII9_3.CPP|EXE 运行时的屏幕截图。它使用的引擎支持彩色光照、纹理映射和恒定着色，场景中有多个光源，带纹理的立方体在不断旋转。

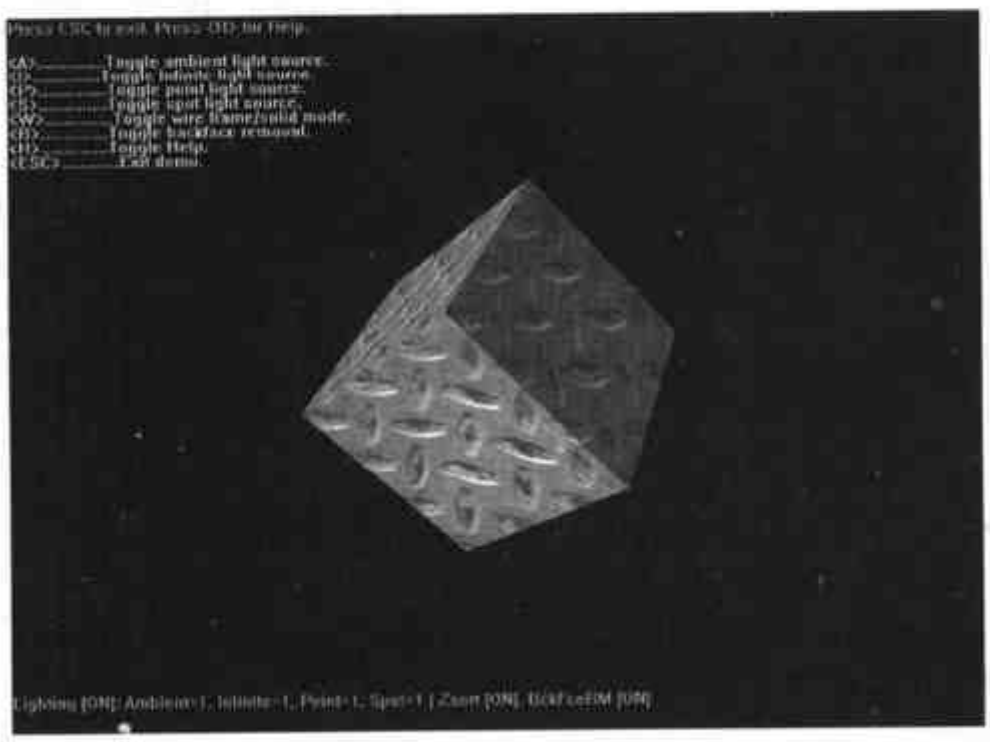


图 9.40　支持彩色光照和纹理映射的 16 位引擎

2. 在 8 位纹理映射函数中添加光照处理功能

支持恒定着色的 8 位纹理映射函数与 16 位版本几乎相同，但内循环除外，而神奇的地方正是内循环。先来看看这个函数的原型：

```
void Draw_Textured_TriangleFS(POLYF4DV2_PTR face,// 指向多边形面的指针
```

```
UCHAR *_dest_buffer,// 指向视频缓存的指针
int mem_pitch)     // 每行占据的字节数（320、640 等）
```

为理解作者使用的算法，请参阅图 9.41。8 位光照存在的问题是，我们使用的是 RGB 光照，而不是单色光照，这就像在汽车快速行驶踩刹车。之所以支持 8 位模式，只是想提高速度。但在这种模式下支持 RGB 光照的原因有两个。首先，您可能需要 8 位的 RGB 光照引擎，例如，您可能是位 GameBoy Advance 程序员；其次，总是可以 RGB 光照计算简化为单色光照计算，但执行相反的转换难得多，因此作者决定将重点放在健壮性而不是速度上。

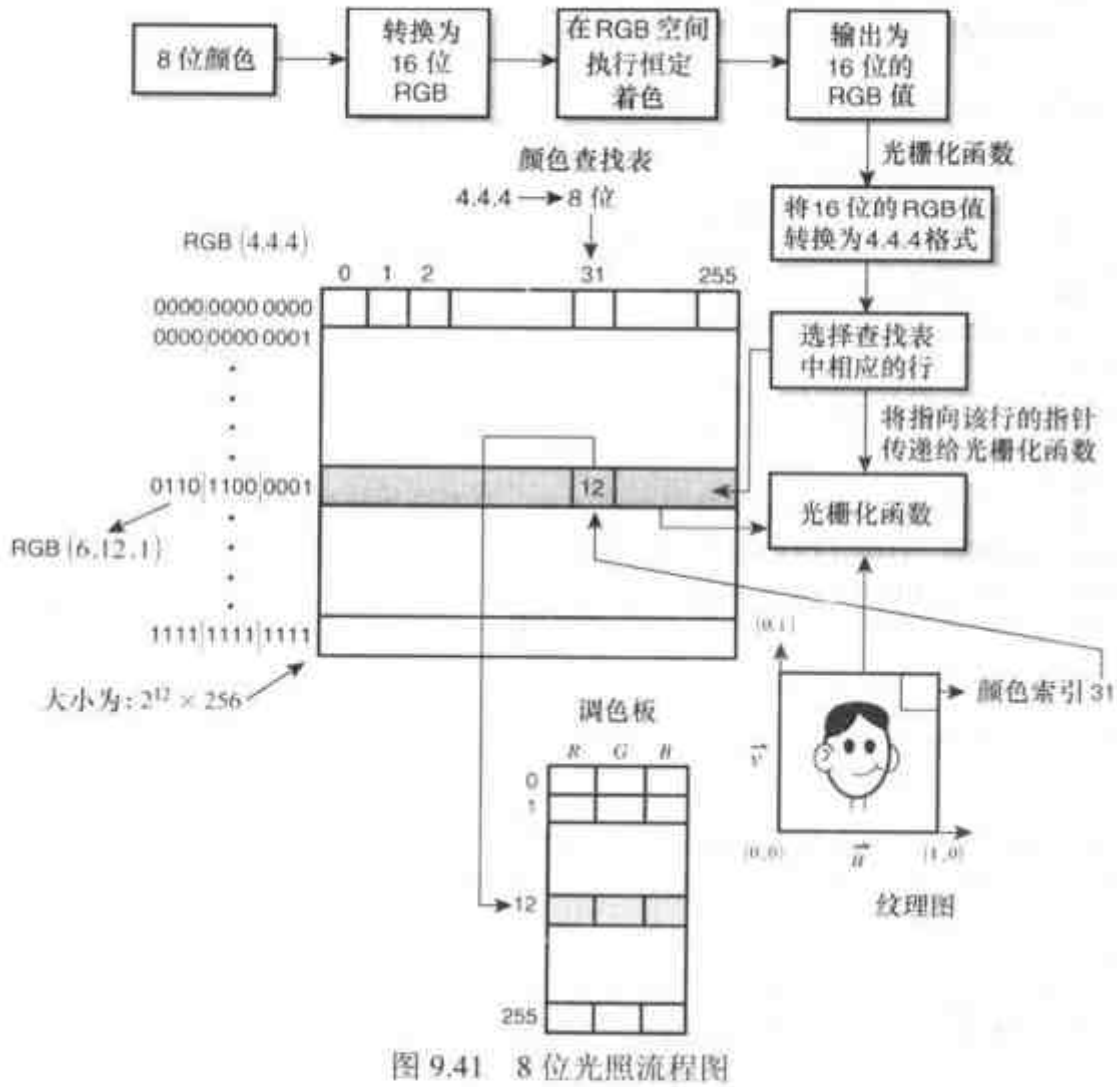


图 9.41　8 位光照流程图

在 8 位模式下，对使用恒定着色且带纹理的多边形进行光栅化时，存在的问题是，不能对颜色进行调制，因为访问 8 位纹理图时，我们处于颜色索引空间中。当然，可以采取这种愚蠢的做法：从纹理图中读取纹素的颜色索引，在调色板中查找其对应的 RGB 值，然后使用 lit_color[0]中的 RGB 颜色对其进行调制，再将结果转换为颜色索引，然后长舒一口气，说在 8 位模式下，速度比 RGB 模式下慢两倍。然而，可以采用更巧妙的方法。

请看图 9.41，作者采取的策略是，创建一个包含任何两种颜色的乘积的查找表。在 16 位模式下，这个查找表将包含 2^{32}（40 亿）个条目。然而，在 8 位模式下，可以采取一些技巧来缩小这个查找表。

在 8 位模式下，只有 256 种颜色。因此可以创建一个 65536 行的查找表，它将涵盖所有 RGB 颜色。也就是说，对于任何一种 5.6.5 格式的 RGB 颜色，我们计算该颜色与索引 0～255 的乘积，然后在调色板中找到与结果最接近的颜色索引。例如，假设要使用 RGB 值(5，10，12)对颜色索引 26 进行调制。前者是 5.6.5 格式的，取值范围为(0...31，0...63，0...31)，将其转换为 8.8.8 格式时，结果为(8，4，8)*(5，10，12) = (40，40，96)。

我们首先提取索引 26 对应的 RGB 值，假设为(90，12，60)，它们为 8.8.8 格式。然后将索引 26 对应的 RGB 值与前面的 RGB 颜色（在 8.8.8 格式下，为(40，40，96)）相乘：

```
最终颜色 = (90, 12, 60) * (40, 40, 96) = (3600, 480, 5760)
```

接下来，需要将各分量缩小到范围 0～255 之内，为此需要除以 255，将结果取整，得到(14，1，22)。

这意味着什么呢？上述 RGB 值是使用 5.6.5 RGB 颜色(5，10，12)对颜色索引 26 进行调制得到的 RGB 颜色。这相当于预先完成了使用一种 RGB 值对一个颜色索引进行光调制的步骤。创建查找表后，便可以根据经过光照处理后的多边形颜色和纹素的颜色索引来查找与调制后的颜色最接近的颜色索引，如下所示：

```
color_lookup[RGBcolor][color_index]
```

其中 RGBcolor 的取值范围为 0～65535，color_index 的取值范围为 0～255。然而，这个表很大——16.7MB。确实需要一个这样大的表吗？答案是否定的。可以采用这样的技巧：将光照引擎可能提供的各种 RGB 值压缩成 4.4.4 格式（红、绿、蓝分量分别有 16 个可能的取值），然后使用它来调制调色板中的 256 个颜色索引。这样，颜色查找表的大小将为 $2^{12} * 256 = 1MB$，这是可以接受的。现在，惟一需要解决的问题是，创建颜色查找表，在使用奔腾 4 CPU 的计算机上，这需要大约 20 秒钟。创建这种查找表的函数如下：

```
int RGB_12_8_Lighting_Table_Builder(LPPALETTEENTRY src_palette,
                    UCHAR rgblookup[4096][256])
{
// 这个函数创建一个查找表，供 8 位纹理映射函数使用
// 需要一个这样的表，对于每种经过光照处理的多边形颜色和纹素颜色
// 表中都包含与它们的调制结果最接近的颜色索引
// 然而，这样的查找表将需要包含 256*65536 个条目，这大了点
// 为缩小查找表的规模，我们将光照处理的多边形颜色压缩为 4.4.4 格式
// 即每个分量只有 16 个不同的取值（共总 12 位）
// 这样查找表的大小将为 256*2^12 = 1MB
// 相对于速度的提高，这是可以接受的
// 另外，这个表是 2 维的，行索引为经过光照处理后的多边形颜色，列索引为纹理的颜色索引
// 这种格式的效率更高，因为查找表是以先行后列的方式存储的
// 同时多边形使用的是恒定着色，
// 因此在纹理映射过程中，经过光照处理的多边形颜色不变，只有纹素的颜色索引不同
// 所以使用 rgblookup[RGBcolor][textel_index]方式访问这个表时，
// 缓存一致性（cache coherence）将非常高
// 最后，调用该函数时，必须提供预先分配好的 rgblookup

// 首先检查指针
if (!src_palette || !rgblookup)
  return(-1);

// 假设使用 RGB 4.4.4 格式，有 4096 个 RGB 值需要计算
for (int rgbindex = 0; rgbindex < 4096; rgbindex++)
  {
// 对于 0～4095 的每个 RGB 颜色值
// 需要将其与颜色 0～255 相乘，然后查找与结果最接近的颜色索引
for (int color_index = 0; color_index < 256; color_index++)
  {
  int curr_index = -1;        // 当前最接近的颜色索引
  long curr_error = INT_MAX; // 当前的距离

  // 从 rgbindex 中提取 R、G、B 值
  int r = (rgbindex >> 8);
  int g = ((rgbindex >> 4) & 0x0f);
  int b = (rgbindex & 0x0f);

  // 将该 RGB 值作为经过光照处理后的多边形颜色
  // 将其乘以纹理颜色（0～255 的颜色索引）
  // 为确保结果在 0～255 的范围内，将其除以 15
  // 这是因为原来的 R、G、B 为 4 位，最大取值为 15
  r = (int)(((float)r *
  (float)src_palette[color_index].peRed) /15);
  g = (int)(((float)g *
  (float)src_palette[color_index].peGreen)/15);
  b = (int)(((float)b *
  (float)src_palette[color_index].peBlue)/15);
  // 在调色板中查找与之最接近的颜色索引
  for (int color_scan = 0; color_scan < 256; color_scan++)
    {
    // 计算与目标颜色之间的距离
    long delta_red  = abs(src_palette[color_scan].peRed  - r);
    long delta_green = abs(src_palette[color_scan].peGreen - g);
    long delta_blue = abs(src_palette[color_scan].peBlue - b);
    long error = (delta_red*delta_red) +
          (delta_green*delta_green) +
          (delta_blue*delta_blue);
```

```
      // 是否更接近
      if (error < curr_error)
        {
        curr_index = color_scan;
        curr_error = error;
        } // end if
      } // end for color_scan

      // 找到最接近的颜色索引，将其存储到查找表中
      rgblookup[rgbindex][color_index] = curr_index;
     } // end for color_index
  } // end for rgbindex
// 成功返回
return(1);
} // end RGB_12_8_Lighting_Table_Builder
```

作者使用一个全局变量来存储这个查找表：

```
UCHAR rgblightlookup[4096][256]; // 颜色查找表
```

在 8 位模式下，必须在加载调色板后像下面这样创建这个查找表，然后再调用 3D 光照函数：

```
RGB_12_8_Lighting_Table_Builder(palette,     // 调色板
                rgblightlookup); // 查找表
```

调用该函数时，将全局查找表和全局调色板作为参数，后者是 T3DLIB1.CPP 中变量 palette。

现在回到 8 位模式下支持恒定着色的纹理映射函数。查找表包含 4096 行和 256 列，每行代表一种 4.4.4 RGB 颜色，其中的元素是同该颜色与 256 种颜色索引的乘积最接近的颜色索引。因此，在 8 位纹理映射函数中，首先需要将一个指针指向查找表中与经过光照处理后的多边形颜色对应的那一行，然后在内循环中，根据纹素的颜色索引在该指针指向的一维表中找到调制结果。全部光调制工作都是通过这个表来完成的。内循环的代码如下：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
{
// 绘制像素
// 读取纹素的颜色
textel = textmap[(ui >> FIXP16_SHIFT) +
       ((vi >> FIXP16_SHIFT) << texture_shift2)];
// 根据纹素颜色在查找表中查找最终的像素颜色，并使用这种颜色绘制像素
screen_ptr[xi] = lightrow444_8[textel];
// 计算下一个像素的纹理坐标 u 和 v
ui+=du;
vi+=dv;
} // end for xi
```

Lightrow444_8[ ]是经过光照处理后的多边形颜色对应的行，其中存储了使用经过光照处理后的多边形颜色（假定纹理下面的多边形的初始颜色为白色）对 256 种颜色索引进行调制的结果。也可以将这种技术用于 16 位光照处理，但查找表将非常大，不过可以使用两个查找表，让 16 位函数版本的内循环与上述代码类似。

有关在 8 位模式下支持纹理映射和恒定着色的演示程序，请参阅 DEMOII9_3_8b.CPP|EXE。它支持彩色光照、纹理映射和恒定着色，场景中有多个光源，带纹理的立方体在不断旋转。

9.8　对 8 位和 16 位模式下优化策略的最后思考

本章提到过很多可优化的地方，介绍过很多优化思想，但其中的 90%都没有实现，因为如果这样做，代码几乎是无法理解的。尽管如此，这里还是要重申其中的一些优化思想，让读者不断地考虑它们。

9.8.1　查找表

有关这种技术三天三夜也讲不完。查找表几乎可用于各个方面，前面的介绍只是蜻蜓点水而已。建议读者检查所有的算法，如果函数的各种输入组合可存储到一个几 KB 到几 MB 的表中，便这样做。当前，

计算机的内存很多，作者使用的计算机有 2GB 的内存，即是游戏使用的查找表有 32~64MB 也不要紧。

数学计算显然是一个可使用查找表的领域，但在 3D 图形学中，还有很多地方也可使用查找表，如颜色映射、纹理映射以及其他相关的操作。另外，可尝试使用级联查找表（cascading lookup table）——让我细细道来。

假设有两个 16 位的输入参数 x 和 y，对于各种 x、y 组合，输出都是一个单字节的值。各种可能的组合数为 2^{32} 个，因此需要使用一个 42 亿字节的查找表——大了点。然而，可以将问题分几个步骤来解决，这样可能首先只需考虑每个操作数的前 8 位，从而可通过一个 64（2^{16}）KB 的表来查找输出，然后再根据输出和余下的 16 位进行另一次表查找。

通过查找表 1 得到的输出可能是 8 位的。8 位的输出加上余下的 16 位数据总共是 24 位，对于每种组合，需要查找一个 1 字节的值，因此查找表为 1600 万字节。这很大，但与 40 亿相比小得多。关键之处在于，有时候可以使用对数运算来分割搜索/计算空间，因此，如果将问题的输入域划分成两块或更多块，便可以通过 2-n 次查找表来解决问题，这样每次涉及的空间将小得多。

另外，还可以缩小输入数据的取值范围，使其更易于处理。例如，在支持恒定着色的 8 位纹理映射函数中，作者将 5.6.5 格式的 RGB 值转换为 4.4.4 格式，并创建一个查找表用于查找最接近的颜色索引。这个表占用的内存不多，但却极大地提高了处理速度。将 5.6.5 格式的 RGB 值转换为 4.4.4 格式时，颜色空间最多缩小了一半，但在 8 位模式下并不明显。这些技巧是读者应该考虑的。

9.8.2 网格的顶点结合性

虽然在 Gouraud 光照模型中，我们没有在着色代码中实现任何顶点结合性，但这确实有待优化。为何在本来只需对立方体的 8 个顶点进行光照计算时，却对 36 个顶点进行光照计算呢？在物体没有被分解为多边形之前，顶点和多边形之间存在一一对应的关系，因为顶点有多个多边形共用——没有重复的顶点。顶点法线、纹理坐标等也是如此。因此，如果对物体而不是渲染列表执行操作（如光照计算）时性能可得到优化，就应这样做——实际上，我们稍后就将这样做。当然，不应对属于背面的多边形执行光照计算，因此不要一味地蛮干，应考虑最大限度地减少工作量。

9.8.3 存储计算结果

前面讨论光照函数时介绍过这种技术。在光照函数中，作者存储了点光源和聚光灯都需要使用的多边形法线。当然，还可以对很多其他计算结果进行存储，那些被重复计算多次的数据都是候选者。为使代码清晰易懂，可首先将用于存储计算结果的变量或字段设置为无穷大，并在使用之前对其进行检测。如果是无穷大，则执行计算，并将结果存储到该变量中。如果不是，则跳过计算代码。例如，下面是一个没有存储计算结果的函数：

```
// 代码块
f1 = sin(x)*cos(y)+zeta
f2 = f1 * cos(y)
f3 = alpha + beta * sin(x)
```

假设由于外部因素的影响，x、zeta、alpha 和 beta 在循环中会发生变化，只有 y 保持不变。因此，可以存储 cos(y)，如下所示：

```
cosy = cos(y)

// 代码块
f1 = sin(x)*cosy+zeta
f2 = f1 * cosy
f3 = alpha + beta * sin(x)
```

在上述代码中，我们知道后面肯定要使用 cos(y)，因此计算它。但如果不能确定后面是否要使用它，则应在需要它时再计算，以免浪费时间。例如，前面的代码块可能不会被调用，因此之前计算 cos(y)毫无用处，只是浪费时间。因此，对于不能确定是否将被使用的值，一种更健壮的策略是，将其设置为无穷大（或其他不可能的标记值），并在需要时才计算它：

```
// 初始化
float cosy = FLT_MAX;
```

```
//. 代码

// ..其他代码

// 仅当需要使用 cosy 时才执行下述代码
if (cosy == FLT_MAX)
  cosy = cos(y);
```

9.8.4　SIMD

SIMD 表示单指令多数据（single instruction multiple data），它是奔腾 III+体系结构的一部分，让您能够对多个数据流执行相同的操作，这意味着可以同时对 4 对数执行乘法运算等。这非常适合我们在 3D 空间中执行的运算。读者可能会问，“这是不是 MMX？”是也不是。MMX（多媒体扩展）是这种技术的第一代，不算太成功。它只是让奔腾处理器能够同时执行 4 种整型运算，但需要占据浮点数寄存器。而 SIMD 确实很管用。

例如，在奔腾 4 处理器中，通过使用内联的 SIMD 指令和/或编译器特定的内部函数，可以将数学运算和 3D 算法的速度提高 400%。这将在第 16 章简要地介绍。这样，对多边形执行光照计算、Gouraud 着色、纹理映射和变换的速度将快得多。有关这方面的详情，请访问 Intel 的网站：

http://www.intel.com

然后，使用关键字“sse”、“sse2”和“simd”进行搜索。

另外，读者还可以下载 Visual C++处理器补丁（processor pack），使 Visual C++编译器能够支持 SSE（流式 SIMD 扩展）/SSE2（流式 SIMD 扩展 2）指令。下载地址如下：

msdn.microsoft.com/vstudio/downloads/ppack/default.asp

9.9　最后的演示程序

现在是介绍引擎演示程序的时候了。作者对前面的坦克程序感到厌烦，因此在介绍过可使之变得非常酷的技术后，作者花了大约 20 个小时的时间创建了一个很酷的坦克演示程序。将坦克演示程序进行升级，使之具备本章介绍的特性很容易，DEMOII9_4.CPP|EXE 是 16 位版本，而 DEMOII9_4_8b.CPP|EXE 是 8 位版本。图 9.42 是该演示程序的屏幕截图，从中可以看到着色效果很不错。这个演示程序与以前的版本基本相同，但场景中的很多物体使用的是 Gouraud 着色，同时由玩家控制的飞行器带纹理且使用的是恒定着色。和以前一样，可以使用表 9.1 所示的键来控制该演示程序。

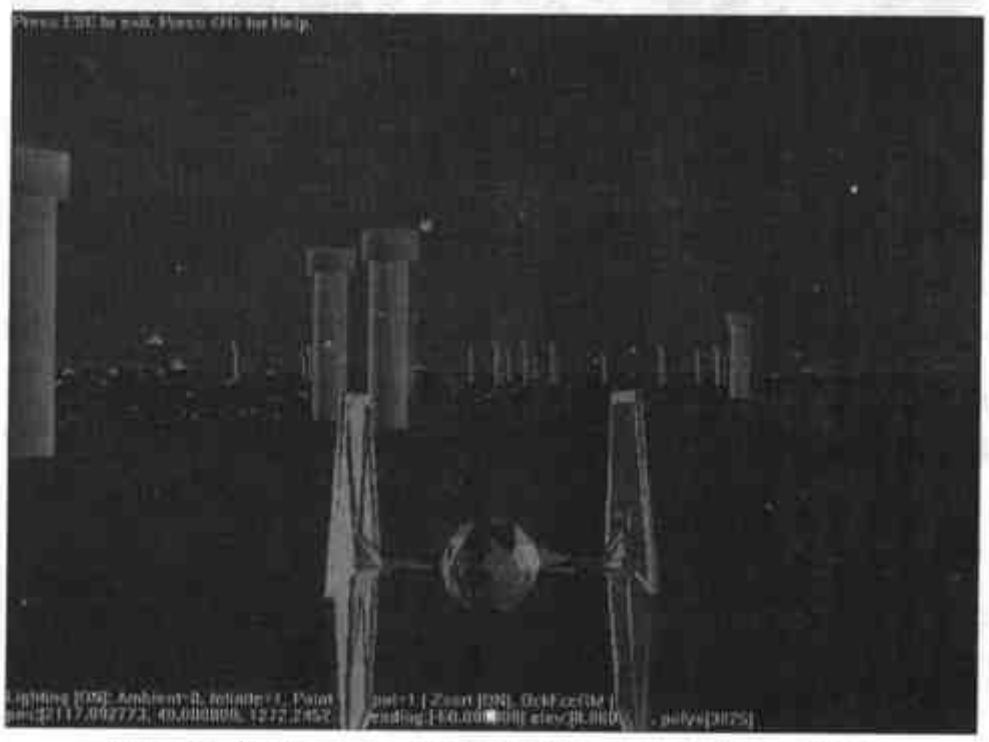


图 9.42　使用 Gouraud 着色和纹理映射的坦克演示程序

表 9.1　　演示程序 DEMOII9_4xx.CPP|EXE 的控制键

键	功能	键	功能
A	开/关环境光源	右箭头键	让玩家向右转
I	开/关无穷远光源	左箭头键	让玩家向左转
P	开/关点光源	上箭头键	玩家前进
S	开/关聚光灯	下箭头键	玩家后退
W	在线框模式和实心模式之间切换	空格键	加速
B	开/关背面消除	H	打开 Help 菜单
L	开/关光照引擎	Esc	退出程序

Raider 3D II

现在介绍更有趣的内容。作者将第 1 章的太空游戏进行了修改，使其中的物体为 3D 实心的，图 9.43 是该演示程序的屏幕截图。它支持到现在为止介绍过的所有特性：彩色光源、纹理映射等。它很简单，只是一个技术演示程序，读者可以添加各种特性。这个演示程序的名为 RAIDERS3D_2.CPP|EXE 和 RAIDERS3D_2B.CPP|EXE（配色方案稍有不同），要编译它们，必须链接模块库 T3DLIB1.CPP|H 到 T3DLIB7.CPP|H。当然，别忘了将所有的多媒体内容复制到可执行文件所在的目录中；另外，所有的纹理图使用的目录名都为：

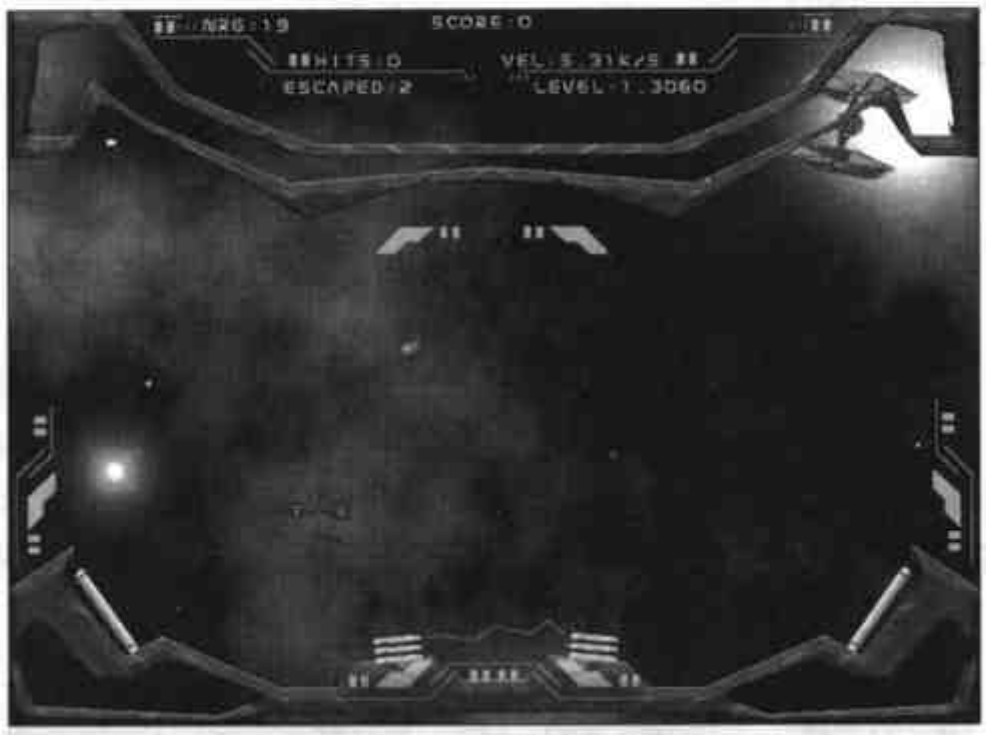


图 9.43　Raider 3D II

```
char texture_path[80] = "./";
```

读者可能想将其设置为“TEXTURE/”或其他目录。如果这样做了，必须将所有的纹理放到.EXE 文件所在目录的相应子目录中。

这个游戏与第 1 章介绍的版本类似，但更先进。使用鼠标来移动瞄准十字线，单击鼠标左键进行射击；要退出游戏，可按下 Esc 键。游戏规则非常简单：玩家必须使用粒子束武器消灭来犯之敌，逃脱的敌人达到 25 个后，游戏将结束。要重玩游戏，只需按下回车键即可。还有一些其他引擎控制方式，例如：

- W——开/关线框模式；
- I——开/关点光源（太阳）；
- A——开/关环境光源。

这个游戏虽然很简单，还是有一些有趣的特性值得讨论，以便读者能够将其加入到自己的游戏中。

1. 星空

为实现星空，玩了一些技巧。我们的引擎不支持星星，因此必须手工渲染它们，并在透视方程中使用与引擎

中相同的视距和窗口大小。在很大程度上说，这里的逻辑很简单，只是在第 1 章的演示程序 RAIDERS3D.CPP|EXE 的基础上进行了拓展。之所以能够这样做，是因为类似于粒子的星星很小，玩家不会注意到是否对它们执行了光照计算和 z 排序。然而，以后确实需要在引擎中添加对粒子系统的支持，这样就不用完这里介绍的花样了。另一方面，作者不喜欢在整个渲染过程中强制物体像粒子一样简单，因此需要考虑使用一种更好的方法。

2. 能量武器

这确实是一种很酷的特效。一开始，作者只是使用直线来表示武器，但转念一想，如果能让粒子（等离子体）能量武器（energy weapon）看起来像闪电，效果一定不错。为实现武器，作者编写了这样的函数：它接受两个端点（将在它们之间绘制能量束）、能量束将被分割成多少段以及最大振幅作为参数。不可思议的是，只经过一次尝试得到的效果就很不错。

3. 碰撞检测和目标跟踪

碰撞检测很容易。基本上，有一个 2D 空间中的瞄准十字线，玩家在屏幕上移动它。但物体在 3D 空间中，因此需要将每艘飞船的包围框投影到 2D 屏幕空间中，然后检测瞄准十字线的中心是否在包围框内。为此，作者计算每艘飞船的包围框，对包围框的 4 个顶点进行投影，然后检测瞄准十字线是否在包围框内。这里涉及的所有知识都在前面介绍过，只是应用它们而已。然而，这种技术的可实现性至关重要，因为它是根据 2D 位置进行 3D 拾取的基础。读者只需牢记：首先将 3D 图像的包围框投影到屏幕空间中，然后进行碰撞检测。

4. 爆炸

爆炸效果没有作者期望的那样激动人心。算法是这样的：将入侵飞船的多边形网格复制到易于处理的爆炸网格中。然而，这里故意没有利用顶点结合性，逐个地复制多边形，因此有些顶点是重复的。这样做旨在方便在爆炸阶段对物体的各个多边形进行移动或变换。在爆炸阶段，以不同的角度和速度将多边形（爆炸碎片）扔出去。其效果不错，但没有类似于位图爆炸（bitmap explosion）的效果。请读者在多边形爆炸的基础上加上位图爆炸——这个游戏需要这种特性，但作者没有时间，而读者有。

5. 字体引擎

正如读者所知，GDI 的速度非常慢，使用它来显示状态或其他调试信息是可行的，但对于显示游戏文本而言，其速度太慢了。因此，作者为这个游戏编写了一个简单的字体引擎，它使用了两个函数，这两个函数位于 RAIDERS3D_2.CPP 中，而不是库模块中。第一个函数如下：

```
int Load_Bitmap_Font(char *fontfile, BOB_PTR font);
```

它加载一个包含模板字体的位图。字体必须包含 64 个字符，它们分成 4 行，每行 16 个字符；每个字符必须是 16×16 像素的。第一个字符为空格字符，请参阅本章对应目录中的文件 tech_char_set_01.bmp。另外，调用上述字体加载函数时，需要传递一个空的 BOB，用于存储字符，每个字符占据一个动画单元格（animation cell）。

第二个函数用于在屏幕的任何位置显示一个字符串。调用该函数时，需要传递包含位图字体的 BOB、坐标 x 和 y、要显示字符串、水平和垂直间隔以及一个可渲染面，如下所示：

```
int Draw_Bitmap_Font_String(BOB_PTR font,
                int x, int y,
                char *string,
                int hpitch, int vpitch,
                LPDIRECTDRAWSURFACE7 dest);
```

建议读者查看这些函数的代码，以了解它们是如何运行的。在下一章，作者可能对它们做一些改进，然后将其加入到库模块中。

6. 游戏中的光照

本章的主题就是光照处理。这个小型演示程序也涉及了这方面的内容。引擎中有 4 个光源：

- 一个环境光源；
- 一个白色太阳，它是一个点光源，位于游戏空间的右上角区域；
- 一颗红色恒星（sun），它是一个点光源，位于游戏左下角区域中用位图表示的恒星附近；

● 一个能量释放器（energy discharge），它是一个点光源，位于玩家的飞船前方的几百个单位处。仅当玩家使用武器时，这个光源才被激活。

提示：请尝试关闭环境光源，以获得更激动人心的光照效果。

7. 最后的说明

这个游戏使用的功能只是引擎的冰山一角。在这个演示程序的基础上再做些工作，可以创建出一个完整的 3D 太空游戏，向游戏玩家兜售。为此，需要增加关卡、目标和故事情节等，但已有的技术已经足够了。如果对这个引擎进行优化，可将速度提高 10～20 倍。请读者尝试运行这个游戏的红色版本（RAIDERS3D_2.EXE）和绿色版本（RAIDERS3D_2B.EXE），它们只是背景不同而已。

9.10 总　　结

本章的篇幅很长，原因在于作者想阐述尽可能多的细节。通过介绍理论知识及其应用，然后编写实现这些功能的代码，我们到达了一个以前从未想过的地方，对这种进步作者很满意。现在，读者即使将本书放到一边，将其中介绍的其他内容抛到脑后，也能够创建出支持 16 位和 8 位模式、纹理映射以及彩色光照的 3D 引擎——这实在是太好了。

下一章的篇幅不会这么长，但进入下一章之前，请读者花些时间来真正理解本章的代码，并尝试运行其中的演示程序。作者没有时间进一步完善引擎，并编写非常棒的演示程序；相对于我们这个粗糙的、未经优化的引擎的功能，本章的演示程序使用的功能只是九牛一毛。相信读者一定能够编写出功能令人惊讶的演示程序。

第 10 章　3D 裁剪

前面一直在避开 3D 裁剪，现在必须面对它。没有办法再避开这个问题——倒不是作者想这样做。本章将介绍 3D 裁剪的理论和实践、为何进行 3D 裁剪及其作用，包括以下主题：

- 裁剪简介；
- 裁剪算法；
- 实现视景体裁剪；
- 地形小议。

10.1　裁 剪 简 介

在本书和《Windows 游戏编程大师技巧》第二版中，多次概要地讨论过裁剪。裁剪是 3D 图形学中最重要的主题，如果没有对几何体进行正确的裁剪，不但无法正确显示它，还可能导致除零异常、内存崩溃以及很多其他棘手的问题。为此，这里再次概述一下各种裁剪以及使用每种裁剪的原因。

10.1.1　物体空间裁剪

物体空间裁剪指的是根据特定的裁剪区域对基本图元组成的几何体进行裁剪。裁剪区域可以是 2D 或 3D 的，重要的是，物体空间裁剪是在数学空间中，根据物体、多边形、图元等的数学表示进行的。物体空间裁剪的优点是，其理论非常清晰：根据 2D 或 3D 视野（view extent）对物体或多边形列表进行裁剪，然后将裁剪后的多边形（在我们的引擎中为三角形）传给 3D 流水线中的下一个阶段。

物体空间裁剪的问题在于细节。在我们的引擎中，处理的总是三角形，必须根据 2D 矩形或 3D 视景体对其进行裁剪，这可能生成顶点数超过 3 个的多边形（非三角形）。换句话说，根据每个裁剪面进行裁剪时，都可能给三角形增加新的顶点，使之成为四边形或其他多边形，因此必须将它们重新分割成三角形。这个处理过程可能并不清晰。如果将引擎设计为支持任何多边形，这将不会成为问题，但我们已经基于只支持三角形进行了很多优化。其次，在物体空间系统中进行裁剪涉及到计算直线之间的交点、直线和平面的交点等。这看似容易，但其实有些棘手，需要解决的问题很多。

1. 物体空间中的 2D 裁剪

本书前面执行过物体空间中的 2D 裁剪，长期以来，我们一直在进行这种裁剪。其基本思想是，将直线或三角形投影到视平面上后，根据矩形视平面对其进行裁剪。本书前面介绍过实现这种裁剪的代码，而《Windows 游戏编程大师技巧》一书对其进行了深入的讨论。我们面临的主要问题是，在 3D 空间中，多边形不但可能穿过近裁剪面，还可能穿过平面 $z=0$，即其顶点的 z 坐标为负，如图 10.1 所示。在这种情况下，仅使用 2D 裁剪不再有效，必须在物体空间中进行更为积极的 3D 裁剪。

2. 物体空间中的 3D 裁剪

如果几何体可以包含任意大小的多边形（如图 10.2 所示），必须根据近裁剪面进行 3D 裁剪。例如，如

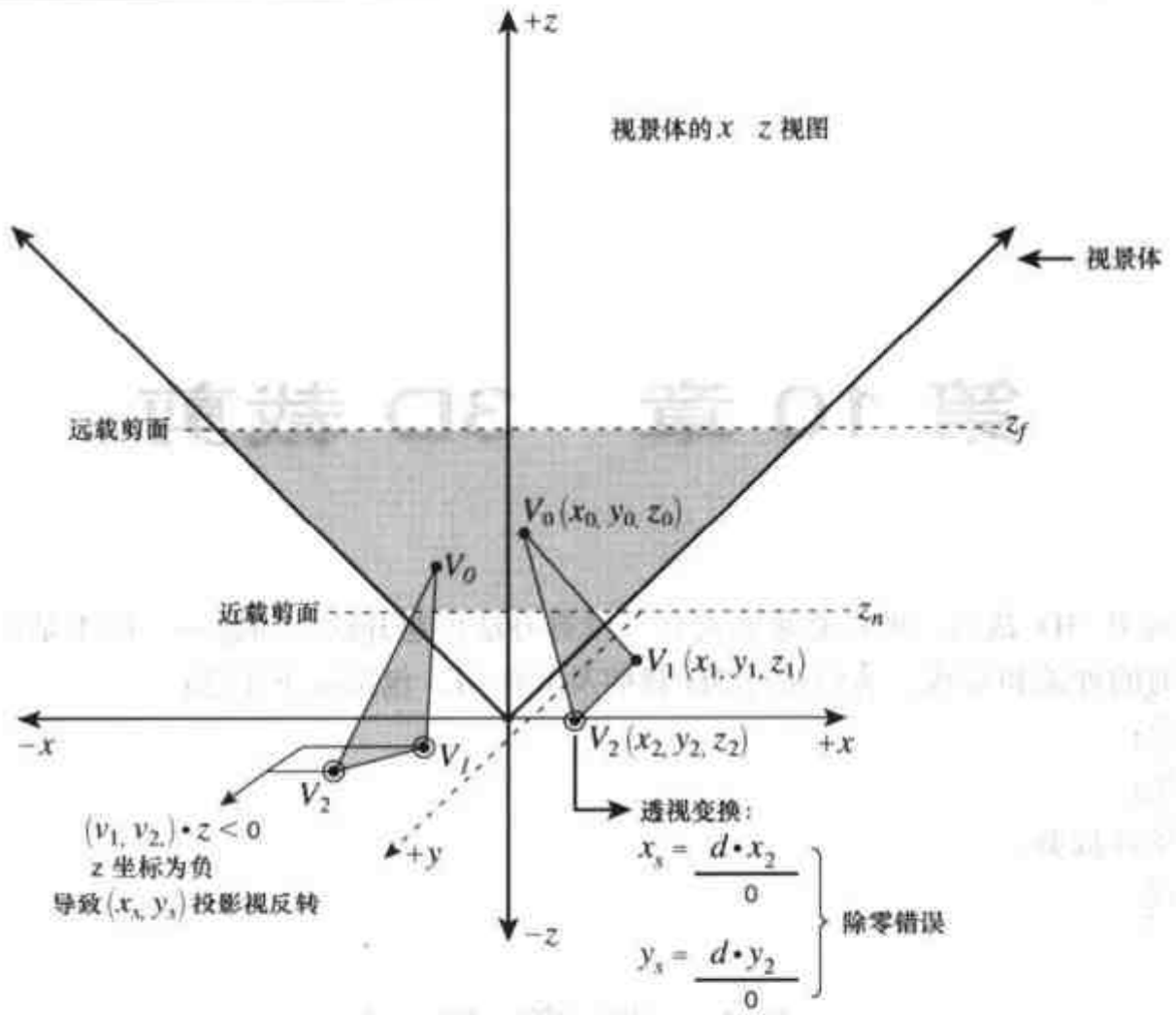


图 10.1　对 z 坐标不大于 0 的顶点进行投影是非常糟糕的

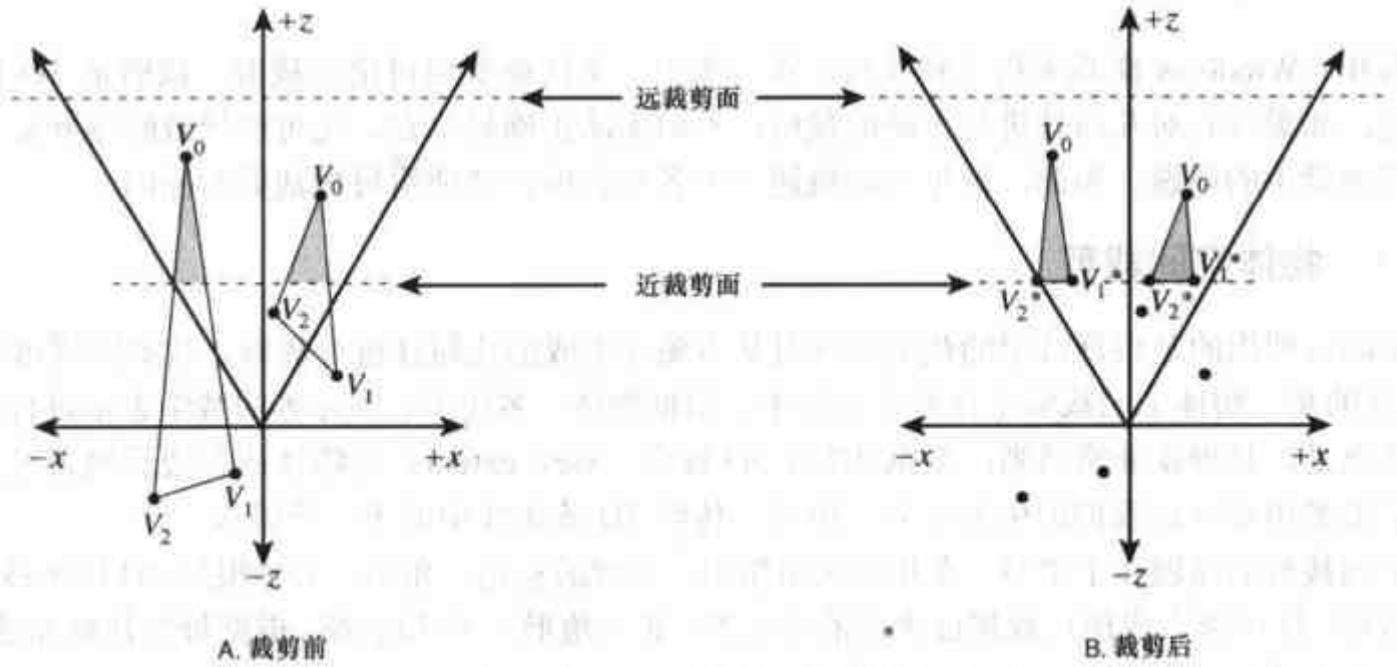


图 10.2　必须对长多边形进行裁剪

果多边形足够小，完全在近裁剪面之外，可以通过简单排除将其剔除，如图 10.3 所示。但如果多边形大小处于中等水平，这种方法将不管用。

至少需要根据近裁剪面对所有多边形进行裁剪，而多边形被裁剪后可能不再是三角形，而是四边形。因此，必须再次将其分割成三角形，并将它们插入到渲染列表中。在最糟糕的情况下，需要使用近裁剪面对所有多边形进行裁剪，这需要计算直线/平面之间的交点。然而，更棘手的是其中涉及的细节。除需要将裁剪后的多边形重新分割为三角形外，还需要对纹理坐标进行裁剪，以便为 Gouraud 着色提供光照信息。另外还需要计算新顶点的法线。涉及的细节很多，稍后将一一介绍。

另一方面，无需使用视景体的其他 5 个面对多边形进行裁剪。如图 10.4 所示，使用远裁剪面对多边形进行裁剪只会浪费时间。通过使用远裁剪面对多边形进行裁剪能得到什么呢？除了浪费时间，什么也得不到，因此我们将排除远裁剪面之外的多边形，而不管其大小如何。如果多边形完全位于远裁剪面之外，我们将剔除它，而不是对其进行裁剪。

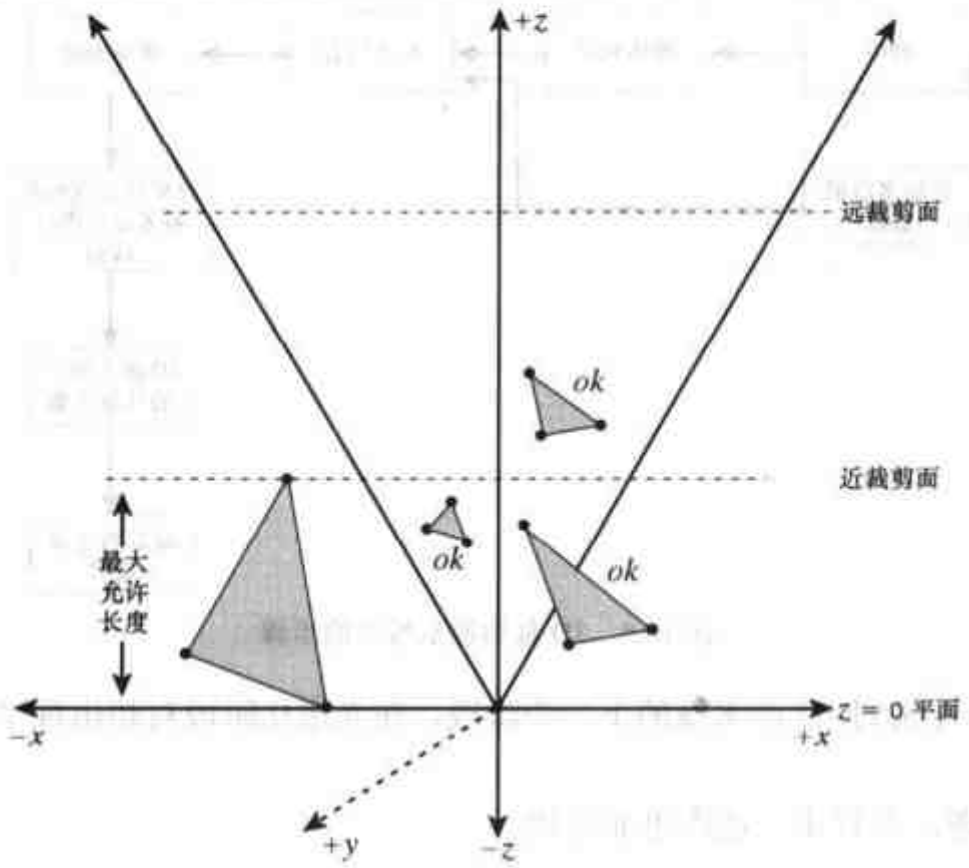


图 10.3　小型多边形可避免裁剪

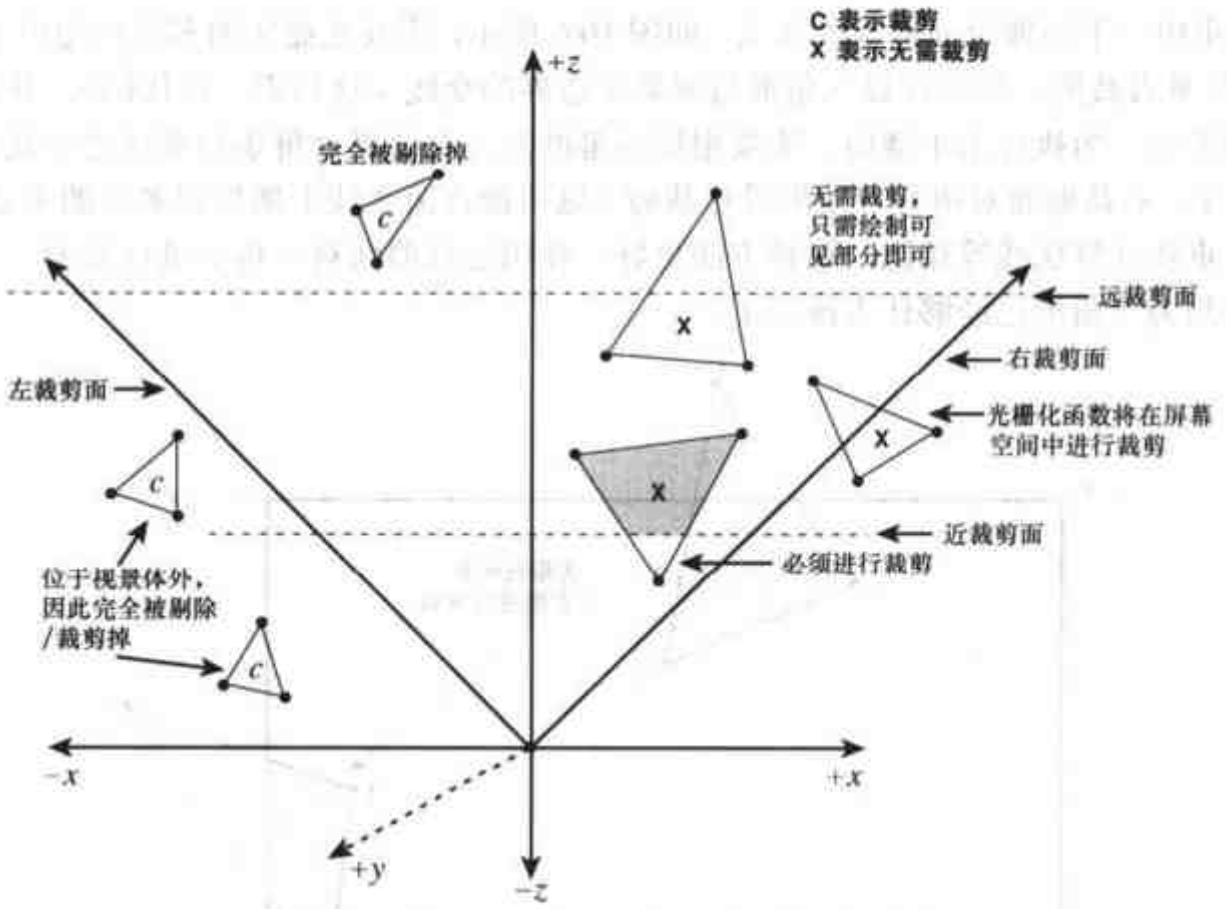


图 10.4　无需使用视景体的所有平面进行 3D 裁剪

同样，使用视景体的上、下、左、右裁剪面对多边形进行裁剪也只会浪费时间。原因在于裁剪操作需要很长的时间。裁剪的目的是什么呢？避免投影后的坐标位于 2D 视平面之外，但这又有什么关系呢？可以在光栅化阶段，在图像空间中对多边形进行裁剪，其速度不比 3D 裁剪慢。另外，根据视景体的上、下、左、右裁剪面进行裁剪时，将导致更多的多边形被加入到流水线中。

我们将采取图 10.5 所示的步骤进行剔除和裁剪：

1．对所有物体执行剔除操作。

2．执行背面消除。

3．根据视景体对所有多边形进行裁剪。只根据近裁剪面对多边形进行完整的裁剪，并将裁剪后的多边形重新分割为三角形。其他裁剪面只用于执行剔除操作（完全拒绝或接受），即如果多边形完全或部分位于上、下、左、右裁剪面定义的半空间内，则接受它；否则将其剔除。

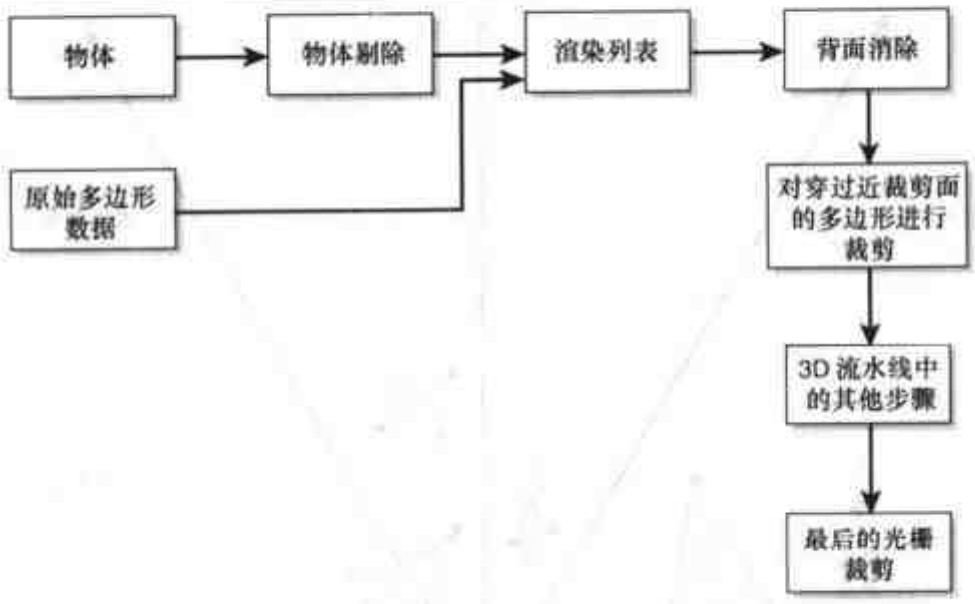


图 10.5　3D 裁剪流水线中的步骤

4. 将余下的多边形传递到 3D 流水线的下一个阶段，在光栅化阶段对超出视平面和屏幕空间边界的三角形进行 2D 裁剪。

这个系统很容易实现，很管用，速度也非常快。

10.1.2　图像空间裁剪

最后，再次重申一下图像空间裁剪的含义。如图 10.6 所示，需要光栅化的多边形超出了视平面/屏幕的 2D 边界。为执行垂直裁剪，只需计算三角形与屏幕上边界的交线（这只需一行代码），并在光栅化循环中从这里开始绘制即可。为执行水平裁剪，需要根据屏幕的左、右边界对每条扫描线进行裁剪，但与使用视景体的上、下、左、右裁剪面对所有三角形进行裁剪（这可能在流水线中增加很多新的多边形）、对其纹理坐标进行裁剪、重新计算法线等相比，这种方法更好。使用近裁剪面对三角形进行裁剪，进而将裁剪后的非三角形重新分割为三角形已经够让人操心了。

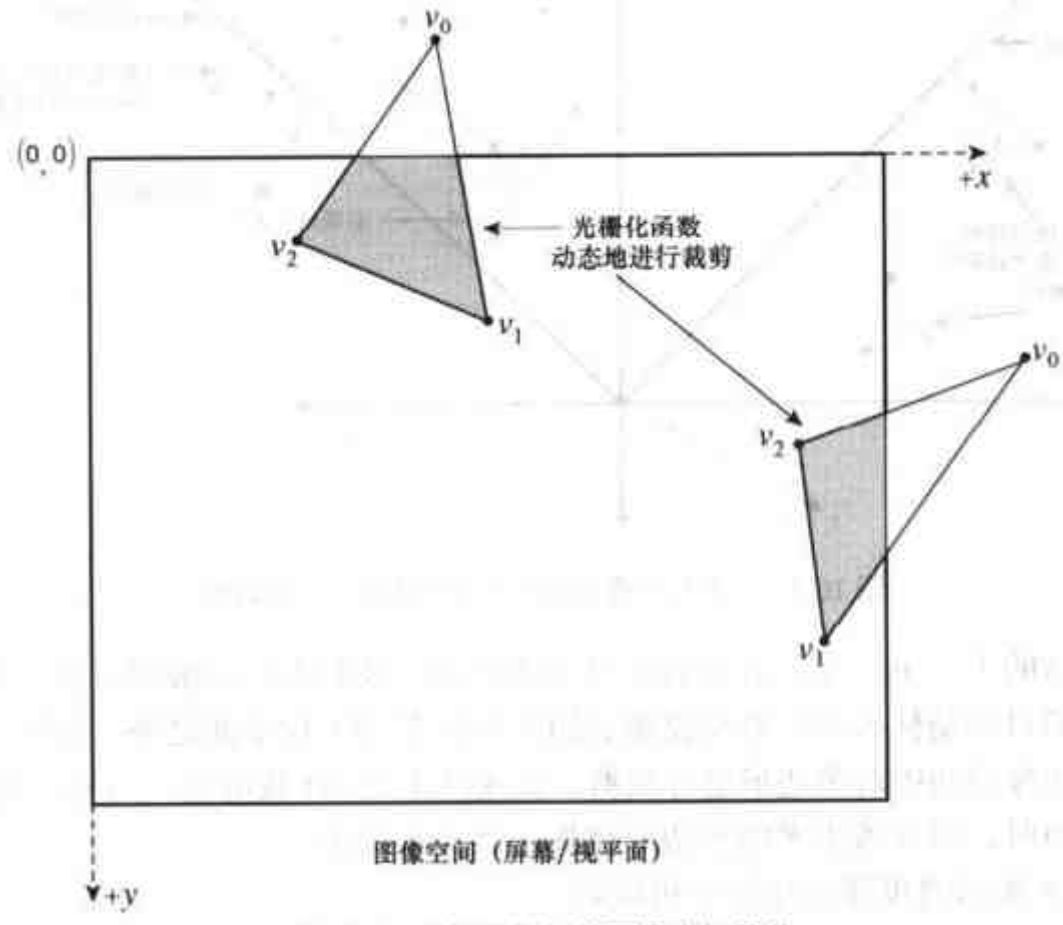


图 10.6　光栅化期间的图像空间裁剪

另一方面，如果采用图像空间裁剪会导致 3D 信息丢失，我们将不会这样做；涉及 z 缓存时，这非常重要。在需要沿多边形边通过插值计算 z 坐标时，必须小心屏幕空间中的操作，在这种情况下，保留顶点的 z 坐标变得至关重要。作者想说的是，最后进行光栅化时，可能发现只有 2D 坐标无法在屏幕空间中进行裁剪。对于这种问题，现在暂时不去管它，以后再考虑。

10.2　裁 剪 算 法

在作者看来，以人名命名裁剪算法是极其可笑的。对于每种裁剪算法，作者在看到它之前都已经想到了——这不是说作者很聪明，而是说对于编写图形算法的人来说，这些算法都是常识性的东西。在计算机图形学领域，以人名给算法命名极其普遍：某人认为她发明了一种新的算法后，将其发表，从此以后，这种算法就以这个人的姓名命名。当然，很多时候，这不是论文作者的意图，她发表了有关该算法的论文后，其他人就将使用其姓名给该算法命名。

裁剪算法过于简单，远未达到将其称为发明的程度。每种算法只是在计算直线交点方面稍微不同，而这些都是常识性的。作者只是根据工作需要编写算法——有时候在重复别人已完成的工作，有时候不是这样。作者想说的是，后面将列出适用的算法，但读者不要认为它们是解决问题的惟一方式。很多时候，当您花几个小时解决某个问题后，却发现已经有人解决过极其类似的问题。对于裁剪，作者希望算法的名称与其内容更为贴切——仅此而已，作者分不清它们是算法名还是律师事务所的名称。

作者认为，要赋予算法以名称，它必须足够复杂，且对该领域中的普罗大众来说并非显而易见的。像二元空间划分法这样的东西应该以发明者的姓名命名，因为它并非显而易见的。然而，当您使用一个多边形对另一个多边形进行裁剪时，在一两个小时内能想到各种线段裁剪、顶点分类、位编码、直线的参数化表示等方法，这些内容正是作者要介绍的。

发完了牢骚，接下来来介绍裁剪算法。作者将从理论的角度讨论一些最流行的裁剪算法，然后根据需要实现一种算法，该算法融合了很多概念，编写它旨在解决问题。下面首先介绍一些有关裁剪的背景知识。

10.2.1　有关裁剪的基本知识

多边形裁剪可简化为线段裁剪，后者又可简化为判断点是否在某个区域内，该区域可以是 2D 的，也可以是 3D 的。图 10.7 从普遍性角度说明了这个问题。场景中的多边形由一组直线/几何体构成，将对场景进行投影，并判断这些实体是否在某种边界内。这可简化为判断各个点是否在边界内。换句话说，所有三角形都由 3 个点组成，这些点定义了 3 条边。虽然这些边使几何体为“实心”或“真实”的，但真正重要的是点。因此，在最低的层次上，只需使用点测试（point testing）判断点(x，y)或(x，y，z)是否在包围区域/体内。

1．2D 空间中的点测试

如图 10.7 所示，给定 2D 空间中的点 p0(x，y)和一个由 xmin、xmax、ymin 和 ymax 定义的矩形区域，如果下述不等式成立，则点位于矩形区域内：

```
{xmin <= x <=xmax}且{ymin <= y <=ymax}
```

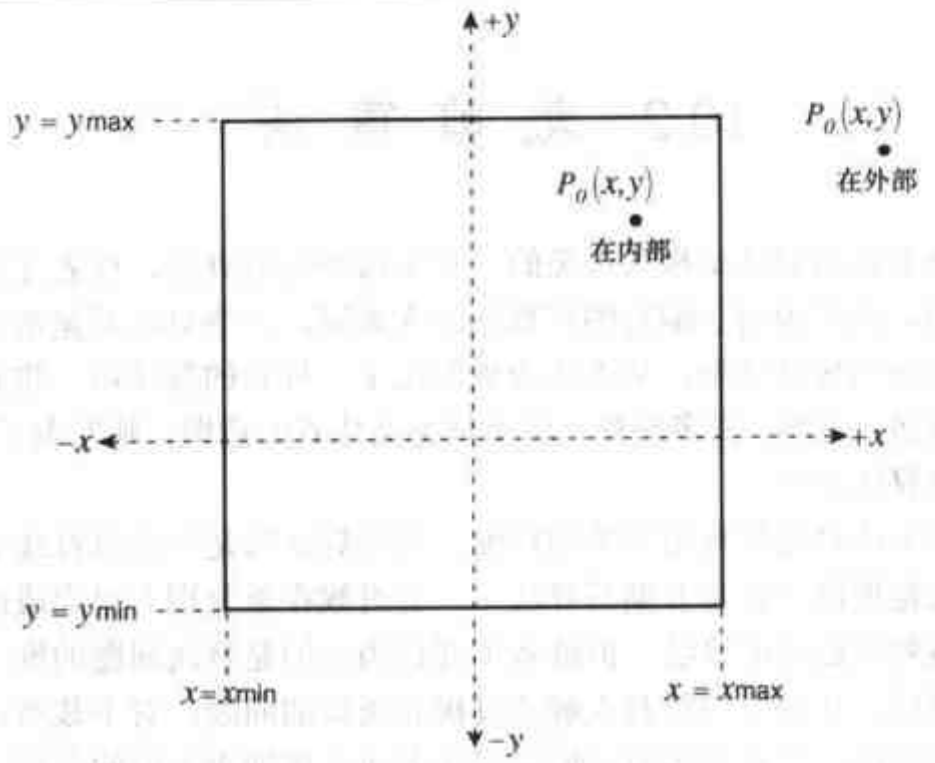
2．3D 空间中的点测试

第 1 种情形　裁剪区域为长方体

如图 10.8 所示，给定 3D 点 p0(x，y，z)和由 xmin、xmax、ymin、ymax、zmin 和 zmax 定义的长方体，当下述不等式成立时，该点在裁剪区域内部：

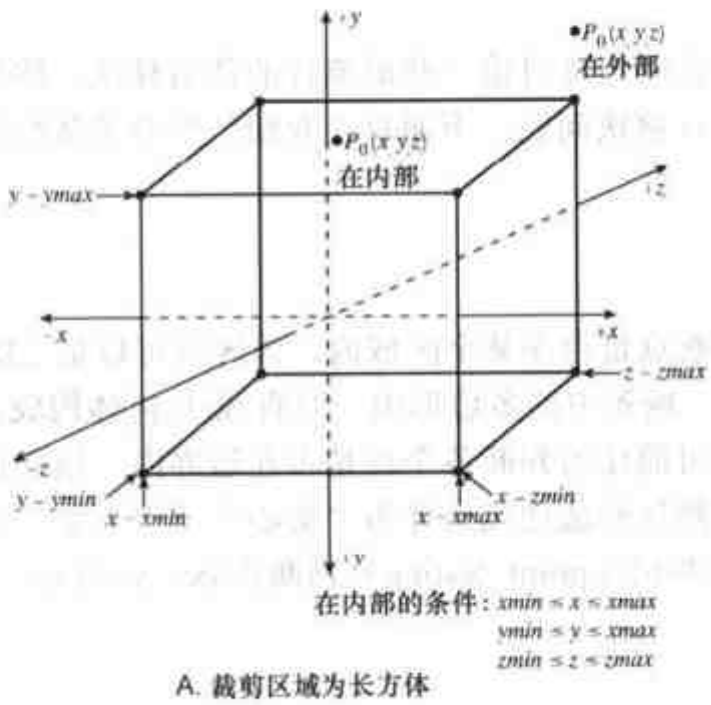
```
{xmin <= x <=xmax}且{ymin <= y <=ymax}且{zmin <= z <=zmax}
```

由于视景体是一个由 6 个平面构成的棱锥台，因此读者可能会问，为何考虑裁剪区域为长方体的情形呢？确实是这样，但读者可能还记得，本书前面讨论投影时指出过，透视投影具有将视景体校正为长方体的功效。这样，相比于需要根据所有的视景体裁剪面进行裁剪，这种裁剪体更简单。另外，没有人说过裁剪都是根据视景体进行的——可能需要使用游戏空间中的一个立方体或矩形对激光束进行裁剪。

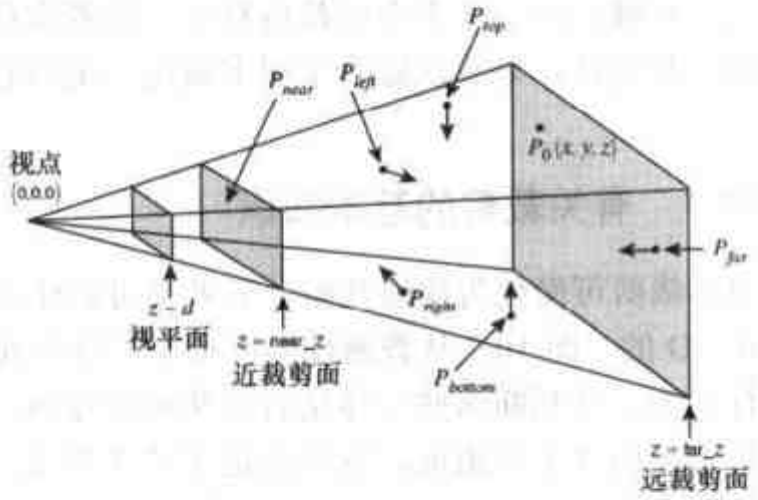


在内部的条件：$xmin \le x \le xmax$
$ymin \le y \le ymax$

图 10.7　2D 空间中的 2 点测试



A. 裁剪区域为长方体



B. 裁剪区域为棱锥台

图 10.8　3D 空间中的点测试

第 2 种情况　视景体为棱锥台

这是我们将进行处理的情形，也是标准情形：视景体由 6 个面构成，必须使用它们对所有几何体进行裁剪。在这种情况下，定义视景体的方法有多种：使用 z 裁剪面的位置以及水平和垂直视野来定义；但不管使用哪种方法，最后都将得到 6 个平面，我们将它们称为 Ptop、Pbottom、Pright、Pleft、Pfar 和 Pnear。另外，假设这些平面都被定义成这样：它们的法线指向视景体的内部。这样，给定点 p0(x，y，z)，我们判断它位于哪个半空间内。正半空间表示点位于裁剪面的内侧，负半空间表示点位于裁剪面外侧（视景体外面）。本书前面介绍过，要判断点位于平面上、正半空间内还是负半空间内，只需将这个点的坐标代入到平面方程中（参见第 4 章和第 5 章）。假设函数 HS(p，pln)计算点 p 和平面 pln 之间的相对关系，+1 表示点位于正半空间内，–1 表示位于负半空间内，0 表示位于平面上。

这样，需要判断下述不等式是否成立：

```
{HS(p0, Ptop) > 0}且{HS(p0, Pbottom) > 0}且
{HS(p0, Pright) > 0}且 {HS(p0, Pleft) > 0} 且
{HS(p0, Pnear) > 0}且{HS(p0, Pfar) > 0}
```

这看似计算量很大，其实并非如此。

3. 线段裁剪基础

知道如何判断点是否在裁剪区域（可以是 2D 或 3D 的）内后，接下来介绍如何对三角形顶点定义的直线/边进行裁剪。为此，需要计算线段与无限长直线和/或有限长线段之间的交点。这在第 4 章和第 5 章介绍过，这里不再深入探讨，只是简要地复习一下。

多边形裁剪可简化为线段裁剪，因此这里只介绍如何使用无限长直线（2D 空间）或平面（3D 空间）对线段进行裁剪。完整的裁剪算法只是重复执行线段裁剪过程，使用每个裁剪边界对构成多边形的每条线段进行裁剪而已。当然，对裁剪算法而言，最重要的是要最大限度地减少计算量，但从很大程度上说，这只是裁剪而已：2D 裁剪、3D 裁剪等。

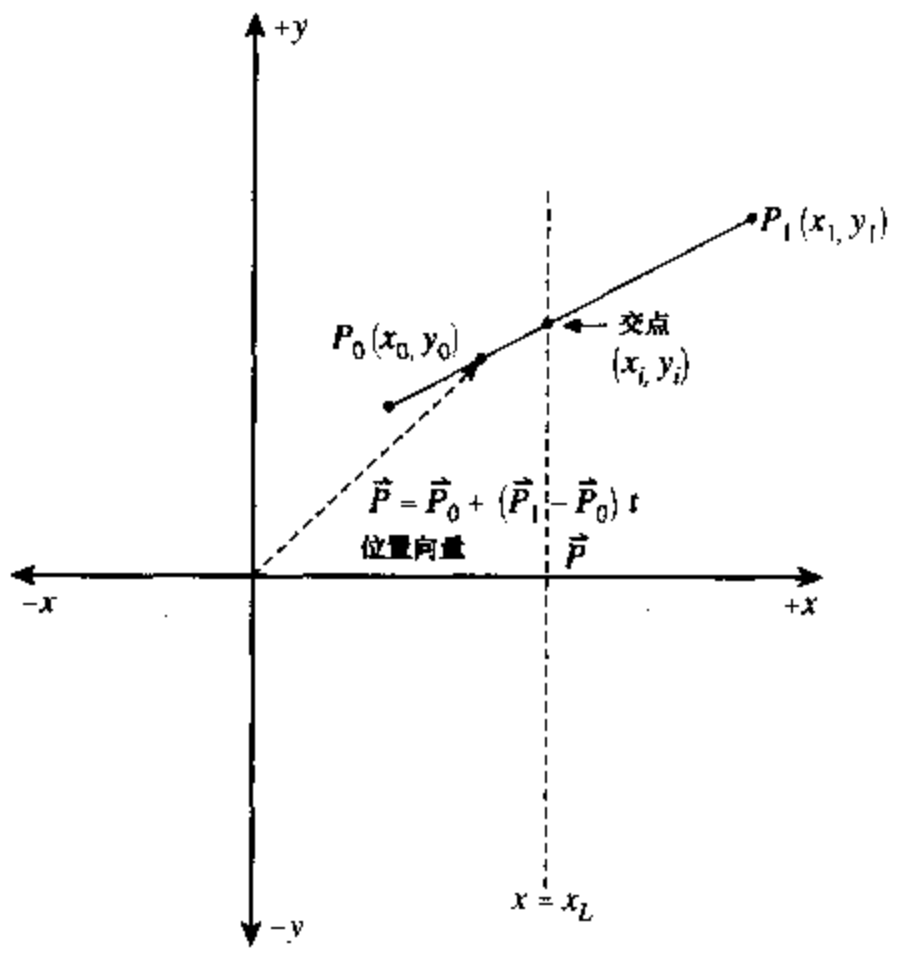


图 10.9　简单的线段裁剪

（1）2D 空间中的线段裁剪

回到线段裁剪的问题，首先介绍 2D 空间的情形。如图 10.9 所示，我们需要使用垂直直线 $x = xl$ 对点 p0(x0, y0) 和 p1(x1, y1) 之间的直线进行裁剪。

注意：在大多数情况下，只需使用水平直线或垂直直线对三角形进行裁剪，因此只需讨论这两种情形。

要对直线进行裁剪，需要将直线转换为参数化形式，然后将垂直直线的 x 坐标代入方程来计算交点。推导过程如下：

使用向量表示法时，线段的方程如下：

```
p = p0 + (p1-p0)*t
```

使用坐标分量表示法时，线段的方程如下：

```
x = x0 + (x1 - x0)*t, {0 <= t <= 1}
y = y0 + (y1 - y0)*t, {0 <= t <= 1}
```

要判断该线段是否与垂直直线相交，可将垂直直线的 x 坐标（这里为 xL）代入到 x 坐标方程的左边，然后求出 t 的值。如果 $0 <= t <= 1$，则线段与直线相交；否则不相交：

```
x = x0 + (x1 - x0)*t
```

将 xL 代入上述方程的左边，结果如下：

```
xL = x0 + (x1 - x0)*t
```

计算 t 的值：

```
(xL - x0)/(x1 - x0) = t
```

如果 t 在 0 到 1 之间，则线段与直线相交。在前面的直线定义中，t 的取值范围为 0～1；当 t 超出这个范围时，仍然在直线上，但不在 p0 到 p1 的线段上。假设前面计算得到的 t 值位于 0～1 之间，只需将其代入下述方程来计算交点的 y 坐标：

```
y = y0 + (y1 - y0)*t
```

这样便得到了线段和直线 x = xL 之间的交点。使用水平线进行裁剪时也很简单，只是使用 y 坐标方程而不是 x 坐标方程来计算 t 值而已。下面来看一下使用水平直线 y = yL 进行裁剪的情况。

将 yL 代入 y 坐标方程的左边，结果如下：

```
yL = y0 + (y1 - y0)*t
```

求出 t 的值，结果如下：

```
(yL - y0)/(y1 - y0) = t
```

同样，对 t 值进行检查。如果 $0 <= t <= 1$，则表明线段与水平直线相交。将 t 值代入 x 坐标方程来计算交点的 x 坐标：

```
x = x0 + (x1 - x0)*t
```

这样，便得到了线段与水平直线之间的交点坐标。

当然，这里需要应用一些常识和拒绝测试。如果端点 **p0** 和 **p1** 重合，则没有必要执行相交测试。同样，试图计算两条水平直线的交点是不合理的。

（2）3D 空间中的线段裁剪

3D 空间中的线段裁剪和 2D 空间中一样简单——这只是一个如何表示平面的问题。到目前为止，还没有使用任何技巧；来看看常规情形：有一个 3D 平面方程和一个参数化 3D 直线方程。图 10.10 定义了这个问题。首先来看平面方程——使用点和法线定义一个平面。

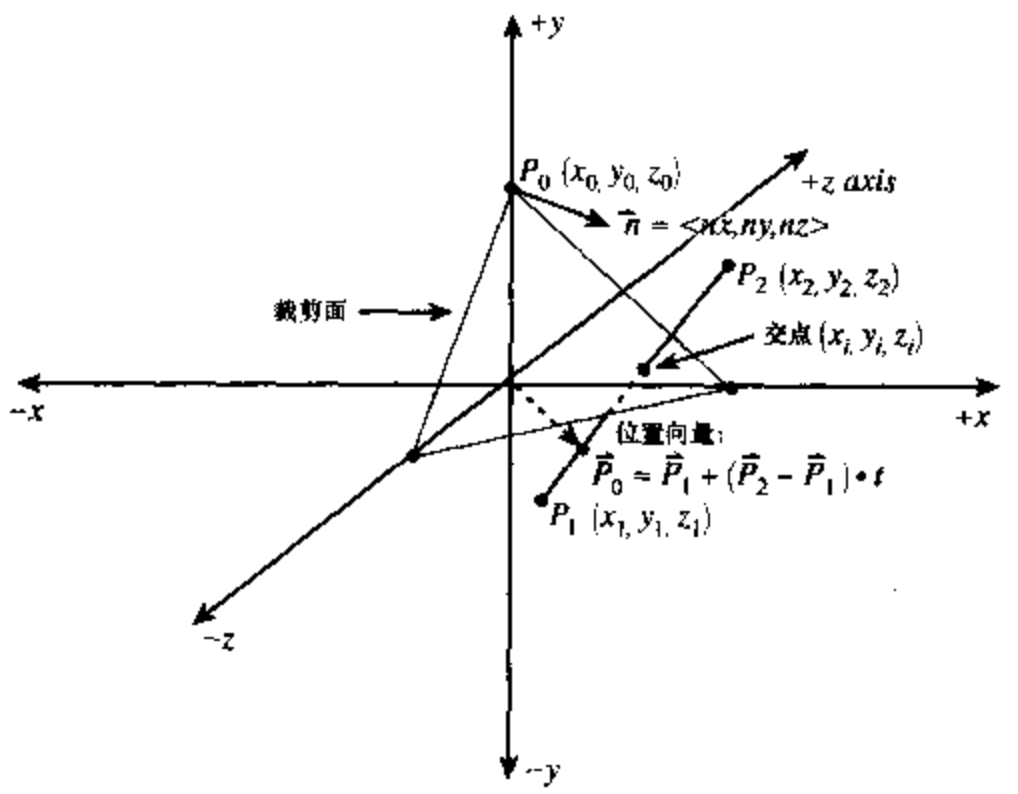


图 10.10　3D 空间中与平面相交的直线

给定平面上的一个点 p0($x0$，$y0$，$z0$)和平面的法线 n(nx，ny，nz)，则平面方程如下：

```
nx*(x - x0) + ny*(y - y0) + nz*(z - z0) = 0
```

使用向量表示时，**p1** 到 **p2** 的直线方程如下：

```
p = p1 + (p2 - p1)*t, {0 <= t <= 1}
```

使用坐标分量表示法时，线段的方程如下：

```
x = x1 + (x2 - x1)*t
y = y1 + (y2 - y1)*t
z = z1 + (z2 - z1)*t
```

现在需要计算交点。同样，这在第 4 章中介绍过。基本方法是，将用参数化直线方程表示的 x、y 和 z 代入平面方程，然后计算出 t 的值。

平面方程如下：

```
nx*(x - x0) + ny*(y - y0) + nz*(z - z0) = 0
```

将 x、y 和 z 代入上述方程，结果如下：

```
nx*(( x1 + (x2 - x1)*t) - x0) +
ny*(( y1 + (y2 - y1)*t) - y0) +
nz*(( z1 + (z2 - z1)*t) - z0) = 0
```

根据上述方程计算 t 的值，然后将其代入参数化直线方程，计算出 x、y 和 z 的值，这便是交点的坐标。

为节省篇幅，这里不再赘述。知道如何计算直线和直线以及直线和平面之间的交点后，裁剪问题就解决了：接下来只需根据具体情况选择一种算法。下面介绍一些比较常见的裁剪算法。

10.2.2　Cohen–Sutherland 裁剪算法

使用 2D 矩形区域或 3D 长方体区域对线段进行裁剪时，Cohen-Sutherland 算法是最流行的方法之一。这种算法由两步组成。

1．端点分类阶段

在分类阶段，将每条 2D 或 3D 线段进行分类，方法是使用位编码标出每个端点位于内部（FALSE）还是外部（TRUE）。图 10.11 说明了在 2D 空间中使用的分类编码方案。

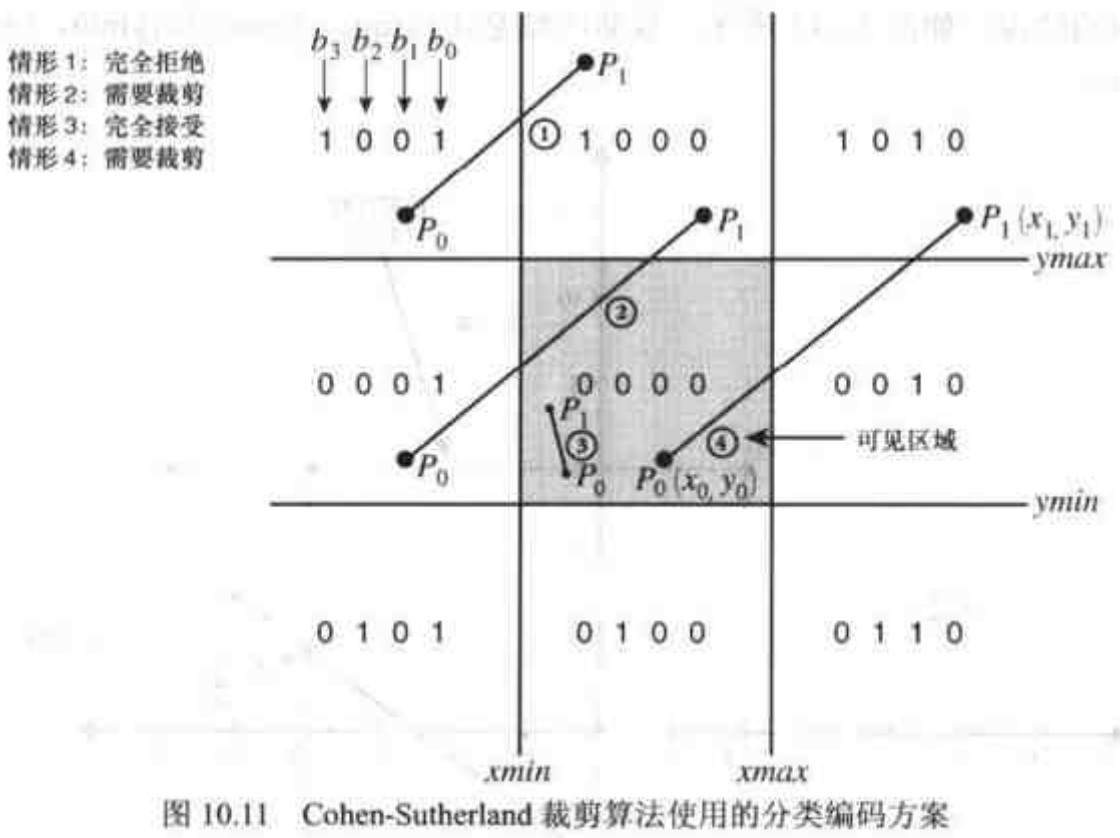


图 10.11　Cohen-Sutherland 裁剪算法使用的分类编码方案

对于 p0(*x*0，*y*0)到 p1(*x*1，*y*1)的线段，对每个端点使用一个 4 位（在 3D 空间为 6 位）的编码进行分类。我们假设端点 p0 的编码为 bitcode0，p1 的编码为 bitcode1。可以采用任何方式对这些位进行编码，表 10.1 给出了一种编码方案——假定 2D 裁剪区域是由左下角(xmin，ymax)和右上角(ymin，ymax)定义的。

对线段的端点进行分类后，进入算法的处理阶段：进行分析和裁剪。

表 10.1　　Cohen-Sutherland 算法采用的位编码方案

位编号	为 TRUE 时的含义	位编号	为 TRUE 时的含义
b3	*y*>ymax，位于上边缘上方	b1	*x*>xmax，位于右边缘右方
b2	*y*<ymin，位于下边缘下方	b0	*x*<xmin，位于左边缘左方

2．处理阶段

在处理阶段，首先判断是否需要完全接受或拒绝线段。这种算法的优点是，通过预先计算避免了编写大量冗余的 if 语句。每个端点都有一个位码，可以使用按位逻辑运算一次性提出多个问题。例如，如果没有分类位码，要判断线段是否在裁剪区域外，需要判断两个端点是否位于裁剪区域的上方、下方、左方和右方，这需要使用大量的 if 语句。但有了分类位码后，这非常简单，可以使用逻辑运算符一次性提出这些问题。下面是完全拒绝/接受的情形。

如果 bitcode0 和 bitcode1 都为 0，则线段 **p0->p1** 位于裁剪区域内，无需进行裁剪，如图 10.11 所示。

下面对位码执行逻辑 AND 运算，并通过检查运算结果来做出判断：

```
bc = (bitcode0 & bitcode1)
```

如果 bc > 0，表明两个端点至少都位于某条边界外，可以完全拒绝该线段，如图 10.11 所示。

如果未通过上述测试，问题将更为复杂——线段与裁剪区域边界有一个或两个交点，需要进行裁剪，如图 10.11 所示。进行裁剪的方法很多，其中之一是使用线段穿过的裁剪区域边界对线段进行裁剪，这将新增一条或两条线段，然后对这些线段应用该算法。为计算交点，可以使用分而治之的算法，也可以使用蛮干算法（这也不错）。作者通常使用这种算法的分类方法，然后在第二个阶段在完全拒绝/接受测试失败后，结合使用其他算法。

10.2.3 Cyrus-Beck/梁友栋-Barsky 裁剪算法

这种算法更为通用，适用于使用（2D 或 3D）凸形区域对参数化线段进行裁剪。它基于这样的假设：使用参数化形式表示线段并使用 t 值而不是坐标(x, y)或(x, y, z)表示交点更自然。这样，可以基于交点的 t 值来执行裁剪，t 值的数量级和取值范围使交点及其属性更为清晰，避免了像 Cohen-Sutherland 算法那样过于纠缠于细节。

来看 2D 空间中的情况，如图 10.12 所示。裁剪区域是由{xmin，xmax}和{ymin，ymax}定义的，2D 线段的参数化方程如下：

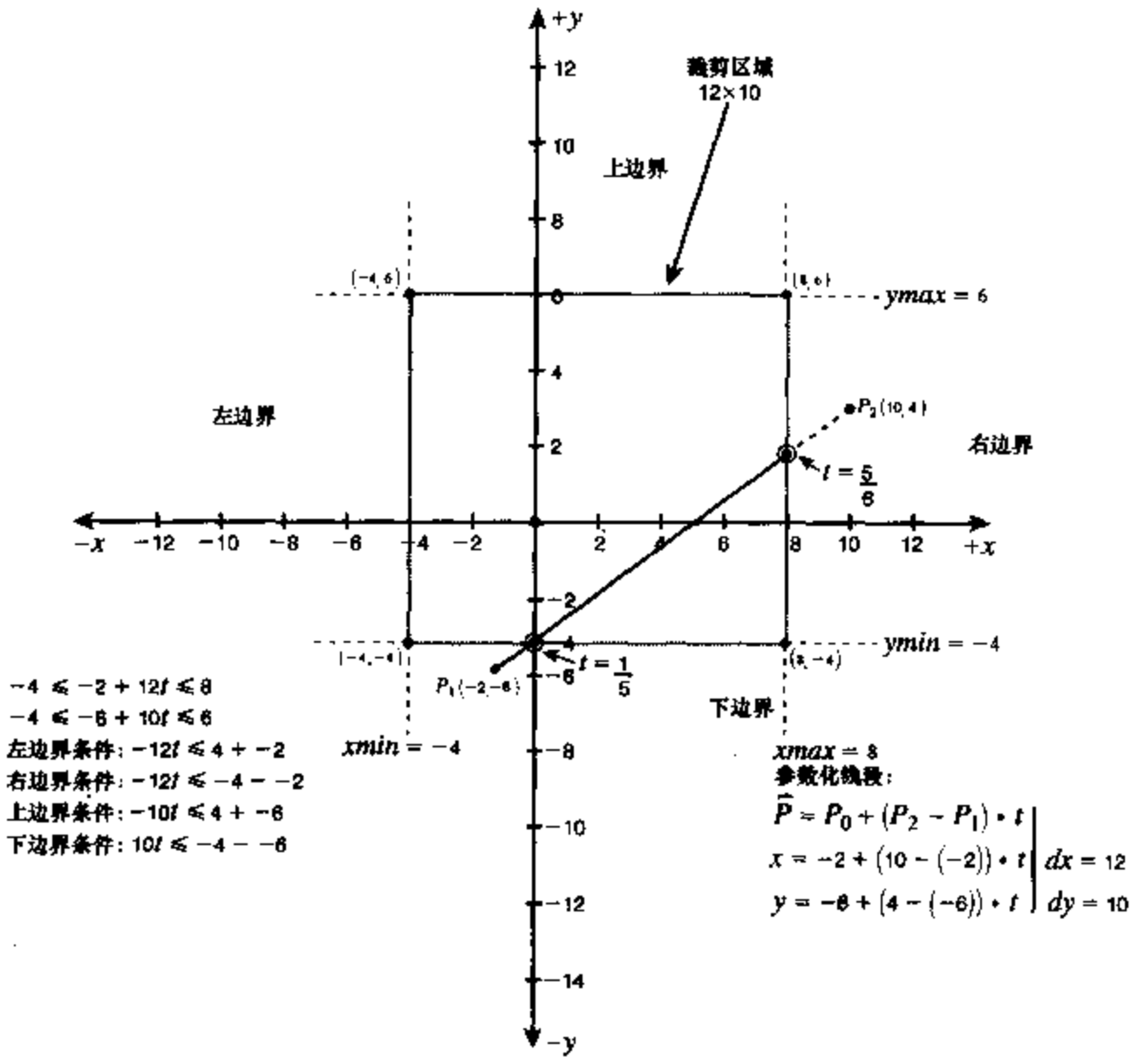


图 10.12 Cyrus-Beck/梁友栋-Barsky 裁剪算法

```
p = p1 + (p2 - p1)*t, {0 <= t <= 1}
```

如果令 dx = x2 − x1，dy = y2 − y1，则以坐标分量表示时，上述方程变为：

```
x = x1 + dx*t
y = y1 + dy*t
```

当下述不等式成立时，线段位于裁剪区域内：

```
xmin <= x1 + dx*t <= xmax
ymin <= y1 + dy*t <= ymax
```

这些不等式可简化为：

```
xmin <= (x1 + dx*t)
```

```
-(x1 + dx*t) >= -xmax
-(y1 + dy*t) >= -ymax
   ymin <= (y1 + dy*t)
```

将 dx*t 和 dy*t 分离出来，结果如下：

```
xmin-x1 <= dx*t
 -dx*t >= -xmax + x1
 -dy*t >= -ymax + y1
   ymin - y1 <= dy*t
```

将上述不等式两边乘以-1，结果如下：

- 左边界不等式（n=0）：-dx*t <= (-xmin + $x1$)；
- 右边界不等式（n=1）：dx*t <= (xmax - $x1$)；
- 上边界不等式（n=2）：dy*t <= (ymax - $y1$)；
- 下边界不等式（n=3）：-dy*t <= (-ymin + $y1$)。

提示：为正确地变换上述不等式，作者花了不少的时间。将不等式方向反转时很容易出错。在两边同时乘以或除以一个负数后，必须将不等式方向反转。

上述不等式可统称为 p_n*t<q_n，其中 n 为不等式的编号，如前所示。这样，将存在下述情形：

情形 1　如果 $p_n = 0$，则线段与边界平行，它们不可能相交；

情形 2　如果 $p_n >= 0$，则线段位于边界内侧；

情形 3　如果 $p_n < 0$，则线段位于边界外侧；

情形 4　如果 $p_n < 0$，则线段穿过边界进入裁剪区域内；

情形 5　如果 $p_n > 0$，则线段穿过边界离开裁剪区域。

只有在情形 4 和情形 5 中，需要将相应的不等式转换为等式，并求解 t 的值：

```
t = qn / pn
```

对于每个不等式都执行上述操作，且令 t 为 r_n，其中 n 为 0、1、2 或 3。

最后，根据线段是通过边界进入裁剪区域（$p_n < 0$）还是离开裁剪区域（$p_n > 0$）将 r_n 分成两组。然后，根据进入裁剪区域的那组 r_n 计算 t1：

```
t1 = max(0, rn)
```

根据离开裁剪区域的那组 r_n 计算 $t2$：

```
t2 = min(1, rn)
```

这意味着什么呢？我们首先判断简单拒绝/接受条件是否成立（即是否属于情形 1、2 或 3——译者注），如果不成立，则属于情形 4 和 5。在这种情况下，必须计算线段和区域边界的交点处的 t 值。为此，需要将前述 4 个不等式视为等式，并计算 t 值，将结果称为 r_n。然后，根据线段是穿过边界进入还是离开裁剪区域将 r_n 分成两组，再分别根据这两组的 r_n 计算出 $t1$ 和 $t2$，它们是线段与边界的交点的 t 值。

如果 $t1 > t2$，表明线段位于裁剪区域外，无需进行裁剪；否则将 $t1$ 和 $t2$ 分别代入到线段的参数化方程中，得到顶点 pi1 和 pi2：

```
pi1 = p1 + (p2 - p1)*t1
pi2 = p1 + (p2 - p1)*t2
```

它们是被裁剪后的线段的端点。这种算法难以理解，但这仅是对人类而言，对计算机来说，这实际上非常简单。该算法很管用，比 Cohen-Sutherland 好得多，因为它是基于参数化线段的，很容易对其进行转换以用于 3D 空间。

算法举例

为让读者深入理解这种算法，来看一个简单的例子。稍微移动一下图 10.12 所示的裁剪区域，使之为：

```
xmin = -4, xmax = 8
```

```
ymin = -4, ymax = 6
```

假定线段保持不变，端点 p1 和 p2 的坐标仍然为(-2，-6)和(10，4)，坐标方程如下：

```
x = -2 + (10 - (-2))*t = -2 + 12*t
y = -6 + (4- (-6))*t = -6 + 10*t
```

因此 $dx=12$，$dy=10$。将这些参数分别代入前面的不等式：

```
-dx*t <= (-xmin + x1)
  dx*t <= (xmax - x1)
dy*t <= (ymax - y1)
  -dy*t <= (-ymin + y1)
```

结果如下：

```
-12*t <= (-(-4) + (-2)) = (2)
  12*t <= (8 - (-2)) = (10)
10*t <= (6 - (-6)) = (12)
  -10*t <= (-(-4) + (-6)) = (-2)
```

因此 p_n、q_n 和 r_n 的值如下：

```
   p0=-12, q0=2, r0=(q0/p0)=-1/6
    p1=12, q1=10, r1=(q1/p1)=5/6
p2=10, q2=12, r2=(q2/p2)=6/5
    p3=-10, q3=-2, r3=(q3/p3)=1/5
```

现在计算 $t1$：

```
t1 = max(0, r_n) = max(0, -1/6, 1/5) = 1/5
```

接下来计算 t2：

```
t2 = min(1, r_n) = min(1, 5/6, 6/5) = 5/6
```

现在检查条件 $t1 > t2$。如果该条件成立，表明线段在裁剪区域外。从图中和计算结果可知，$t1 < t2$，因此需要将 $t1$ 和 $t2$ 代入参数化线段方程中，得到裁剪后的线段。将 $t1 = 1/5$ 代入方程，结果如下：

```
x1' = -2 + 12*(1/5) = -2 + 2.4 = .4
y1' = -6 + 10*(1/5) = -6 + 10/6 = -4.0
```

将 t2 = 5/6 代入方程，结果如下：

```
x2' = -2 + 12*(5/6) = -2 + 10 = 8
y2' = -6 + 10*(5/6) = -6 + 50/6 = 2.3
```

因此，裁剪后的线段端点为 p0'(0.4，-4.0) '和 p1'(8.0，2.3) '。从图 10.12 可知，这是正确的。

该算法以清晰的方式对一些信息进行分类和组织。具有讽刺意味的是，不知道该算法的程序员编写的很多裁剪函数采用了与该算法相同的信息流程，这表明这种算法确实不错。

这里不打算实现该算法，本书后面讨论优化时，将结合使用 Cohen-Sutherland 算法和本算法进行 2D 裁剪。那么，该算法可用于 3D 裁剪吗？当然。在 3D 空间中，该算法的步骤相同，只是需要使用不等式来判断点位于裁剪面的正半空间还是负半空间中，因此需要使用点积来进行半空间测试。

10.2.4 Weiler-Atherton 裁剪算法

这是本章将介绍的最后一种裁剪算法，它更符合这样一种理念：多边形是由顶点构成的，而顶点要么在裁剪区域内，要么在裁剪区域外。该算法用于使用任何一个多边形对另一个多边形进行裁剪。根据上述规则，如果两个顶点分别位于裁剪区域内和裁剪区域外，则它们定义的多边形边要么是进入裁剪区域，要么是离开裁剪区域。如图 10.13 所示，要裁剪的多边形称为受裁多边形（subject polygon），定义裁剪区域的多边形称为裁剪多边形（clipping polygon）。

要使用这种裁剪算法，必须用顶点列表定义这些多边形，将顶点存储在数组或链表中，并按一致的顺序排列它们——我们假设两个多边形的顶点都顺时针方向排列。该算法从受裁多边形的顶点 0 开始，计算以它为起点的边与裁剪多边形每条边的交点。如果找到了这样的交点，将其插入到受裁多边形和裁剪多边

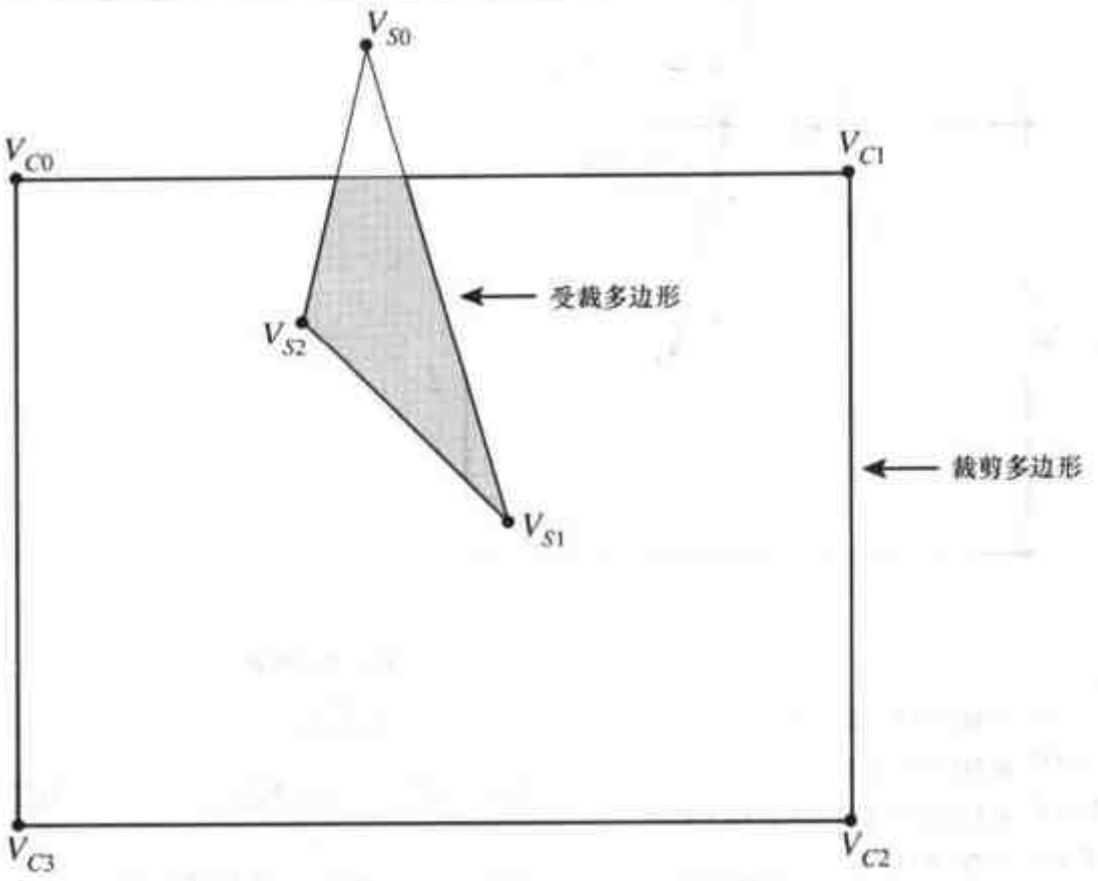


图 10.13　Weiler-Atherton 裁剪算法

形的顶点列表中（位于交点所在多边形边的两个顶点之间），并使用进入/离开标记指出是进入还是离开裁剪区域时形成的交点。遍历完受裁多边形，并将所有交点插入到顶点列表中后，执行下述步骤来生成裁剪后的多边形：

1. 在受裁多边形的顶点列表中找到第一个进入裁剪区域的交点，将其作为裁剪后的多边形的第一个顶点。

2. 遍历受裁多边形的顶点列表，直到找到下一个交点。在此过程中，将访问过的每个顶点都加入到裁剪后的多边形中。根据定义，该交点必定是离开裁剪区域时形成的。

3. 在裁剪多边形的顶点列表中进行遍历，找到与第 2 步中相同的交点，让两个顶点列表同步。

4. 继续遍历裁剪多边形的顶点列表，直到找到下一个交点，并将在此期间访问的每个顶点加入到裁剪后的多边形中。该交点是在进入裁剪区域时形成的。

5. 如果第 4 步找到的交点不是裁剪后的多边形的第一个顶点，则回到第 2 步（旨在让裁剪后的多边形是闭合的）。

6. 如果第 4 步找到的交点是裁剪后的多边形的第一个顶点，则裁剪后的多边形将是闭合的，但工作可能还未完成。如果裁剪后的多边形包含受裁多边形顶点列表中所有进入裁剪区域的交点，工作便完成了（可使用计算器或标记来判断）；否则，继续遍历受裁多边形的顶点列表，找到下一个进入裁剪区域的交点后跳到第 2 步。

上述算法很不错，适用于使用任何一个多边形对另一个多边形进行裁剪。图 10.14 是一个简单的例子，说明了该算法的流程。

作者不喜欢预先处理多边形顶点列表，然后遍历顶点列表多次的做法。别忘了，我们的引擎处理的是三角形，通常承受不起复杂算法的开销。在大多数情况下，我们将基于上述思想编写一种速度非常快的裁剪算法。下面的算法基于 Weiler-Atherton 算法中进入和离开裁剪区域的概念。

首先判断三角形的第一个顶点在裁剪区域内还是裁剪区域外。如果在裁剪区域内，则输出它；否则，遍历到下一个顶点。如果三角形边的起点和终点的状态不同（一个在裁剪区域内，一个在裁剪区域外），则计算这条边与裁剪区域边界的交点，并输出它。基本上，这只是遍历多边形的每条边，如果其起点位于裁剪区域中，则继续遍历，在离开裁剪区域时输出交点；然后继续遍历，在再次进入裁剪区域时输出交点，并继续遍历。这实际上是 Weiler-Atherton 算法的简化版本，但速度更快。在大多数时候，作者都使用这种算法，因为在知道裁剪多边形和受裁多边形都很简单的情况下，为其创建复杂的顶点列表用于裁剪有些小题大做。图 10.15 通过一个简单的例子说明了这种算法。

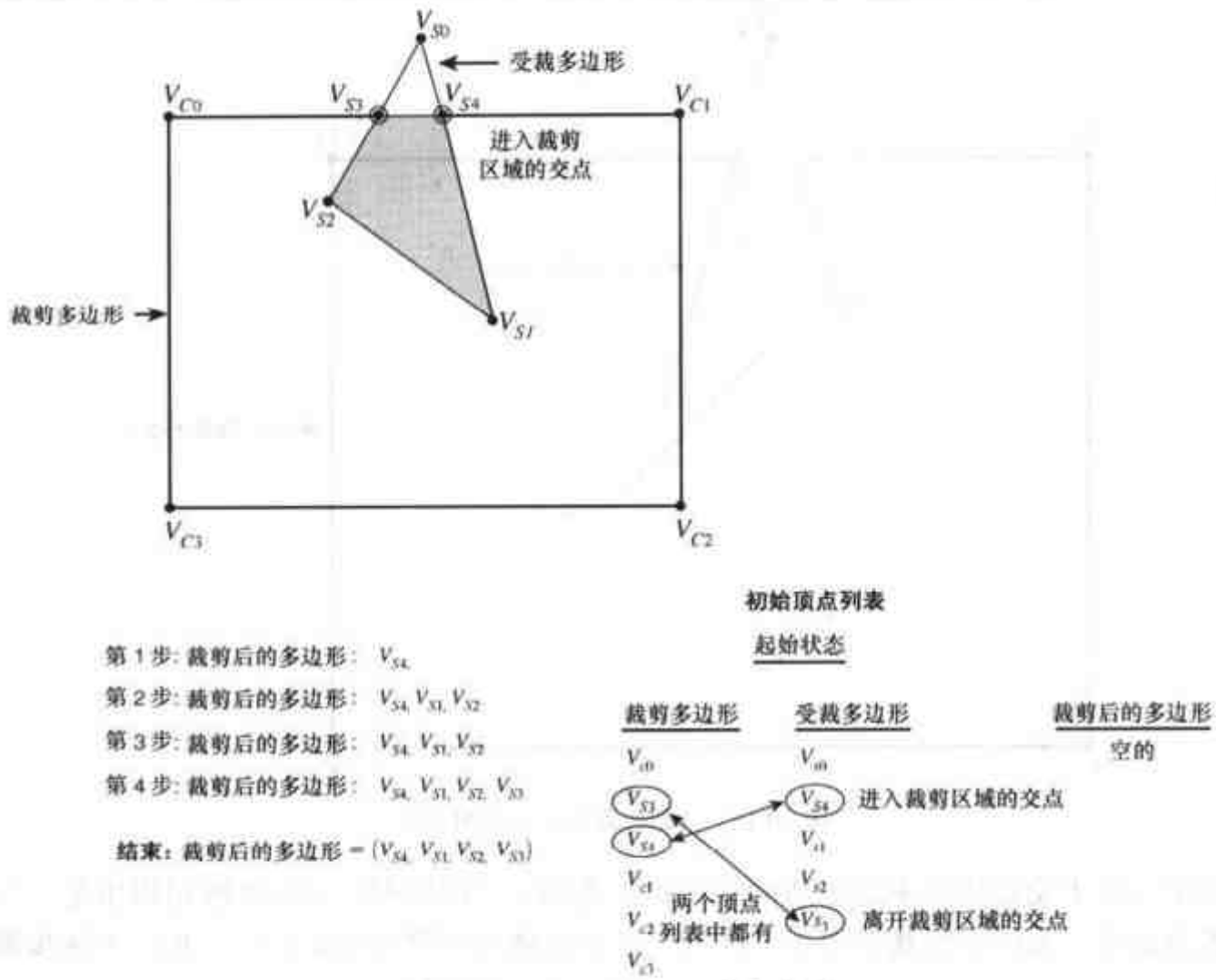


图 10.14　Weiler-Atherton 算法举例

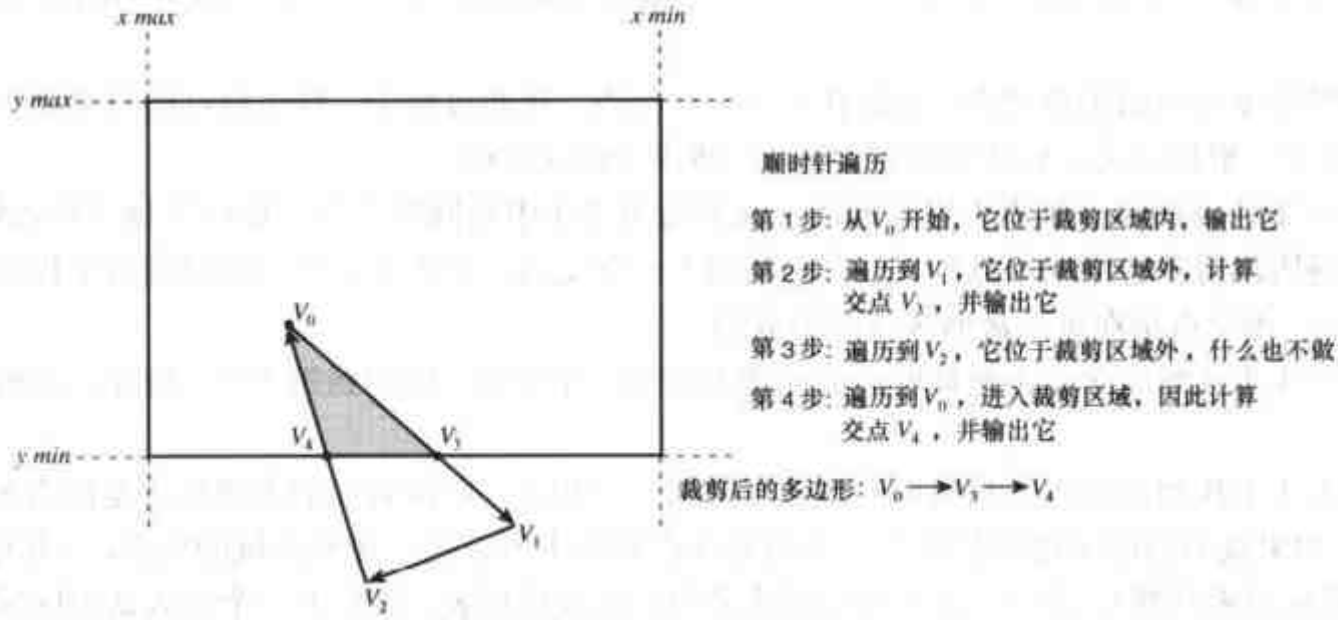


图 10.15　简化的 Weiler-Atherton 算法

10.2.5　深入学习裁剪算法

至此，读者知道如何进行裁剪了。说真的，这并不难——只需使用一种算法或多种算法的组合找出多边形与 2D 或 3D 裁剪边界的交点即可。接下来我们将在引擎中实现 3D 裁剪功能。如果读者想更深入地了解裁剪算法，有一些不错的图书可供参考。它们从学术性的角度讨论了裁剪，虽然对实现实时游戏算法来说用处不大，但它们全面而详细地讨论了算法。下面是作者爱读的一些计算机图形学方面的图书：

- Foley、van Dam、Feiner 和 Hughes 编写的 *Computer Graphics: Principles and Practice*（Addison-Wesley 出版）。
- Foley、van Dam、Feiner、Hughes 和 Philips 编写的 *Introduction to Computer Graphics*（Addison-Wesley 出版）。
- Alan H. Watt 编写的 *3D Computer Graphics*（Addison-Wesley 出版）。
- David F. Rogers 编写的 *Procedural Elements for Computer Graphics*（麦格劳希尔出版）。

10.3　实现视景体裁剪

现在是在 3D 引擎中添加裁剪功能的时候了。基本上，需要在流水线的某个地方根据 3D 视景体对多边形进行裁剪。正如前面指出的，我们的进攻计划是，对多边形进行 3D 裁剪，但尽可能减少分割裁剪后三角形的工作量，只执行必不可少的 3D 裁剪。

那么，在什么情况下必须进行裁剪呢？仅当多边形穿过近裁剪面甚至平面 z=0 时，才必须进行裁剪。如果多边形包含 z 坐标小于或等于 0 的点，对其进行投影将导致除零运算或投影反转，这是不能接受的。

到目前为止，我们一直将视景体的近裁剪面设置为离投影平面和平面 z=0 足够远，以避免多边形穿过它们，因为我们将那些离视点和平面 z=0 太近的物体剔除掉了。然而，现在需要将物体级剔除抛到一边，更多地从多边形列表的角度考虑问题。这些多边形可能引发问题，因为它们可能位于任何地方。

我们将编写一个混合型裁剪/剔除函数，它根据视景体对渲染列表中的多边形进行裁剪——但实际上只根据近裁剪面对顶点进行裁剪。对于穿过上、下、左、右或远裁剪面的多边形，则不去管它。对于完全在这些裁剪面之外的多边形，则将其剔除。当然，被投影后，多边形可能超出 2D 视平面的边界，但光栅化函数将对其进行裁剪，而不会带来任何问题。正如以前指出的，与根据视景体的所有 6 个面对每个多边形进行裁剪相比，这种策略使裁剪更简单，速度更快。根据其他 5 个裁剪面进行裁剪将浪费大量时间。对多边形进行分割、重新计算法线和纹理坐标毫无意义，因为无论如何，在光栅化函数中仍需要完成这些该死的工作。仅当多边形非常长，穿过了近裁剪面，可能在投影期间导致失真或数学运算错误时，才需要进行裁剪。我们采用的方法的优点是，不但将消除背面，还将剔除不在视景体内的多边形。这样，将比以前消除更多的几何体，获得更高的性能。

可在 3D 流水线中的很多地方进行 3D 裁剪：世界坐标空间、相机空间或透视投影空间；这些方法各有优缺点。例如，在世界坐标空间进行裁剪时，可避免将所有几何体都变换为相机坐标，但视景体位于视点处，朝向是任意的；因此裁剪和剔除方法必须更通用。在另一方面，对多边形进行世界坐标到相机坐标变换后，它们的位置是相对于位于原点处的相机的，视景体的各个面都平行或垂直于坐标轴，这使得裁剪更为容易。这以对所有几何体进行相机坐标变换为代价，使裁剪/剔除更容易。如果在执行透视投影后进行裁剪（这有些棘手），几何体都被归一化，而视景体也变成了规范的透视视景体——长方体。换句话说，如果执行透视变换后再进行裁剪，将根据长方体而不是棱锥台进行裁剪——这在有关数学和 3D 投影的章节中提过多次。

那么，该如何办呢？将执行透视投影后进行裁剪的方法抛在脑后吧——等到那时候再进行裁剪实在太糟糕，将对那些本不需要进行光照处理的多边形执行光照计算，同时会在流水线中增加 200%的不会被渲染的多边形，这实在是浪费。可以在世界坐标空间中进行裁剪，但作者不喜欢对视景体进行变换后再进行裁剪的想法。另外，对于被分割的多边形，也需要进行变换。假设场景中有 1000 个多边形，由于裁剪和分割，最后将得到大约 1200 个多边形；这样，将必须对 1200 个多边形进行相机变换。另外，在世界坐标空间中对多边形进行裁剪测试比在相机空间中更复杂，在相机空间中，相机位于原点，观察方向与 z 轴平行。

另一方面，假设在世界坐标空间进行裁剪可消除大约一半的多边形，则只需要对 500 个多边形进行相机变换。上述两个例子表明，这两种方法各有优缺点。作者决定在执行相机变换后进行裁剪/剔除，因为这样更容易理解。另外作者不相信在世界坐标空间中执行裁剪/剔除时可消除那么多多边形。在本书最后介绍优化时，可能尝试采用世界坐标空间裁剪。

总之，我们将在执行相机变换后进行裁剪/剔除：剔除完全在视景体外的多边形，同时只对穿过近裁剪面的多边形进行裁剪。对于穿过上、下、左、右裁剪面的多边形，则不去管它，因为光栅化函数将在图像空间对其进行裁剪。图 10.16 是加入裁剪函数后的流水线。

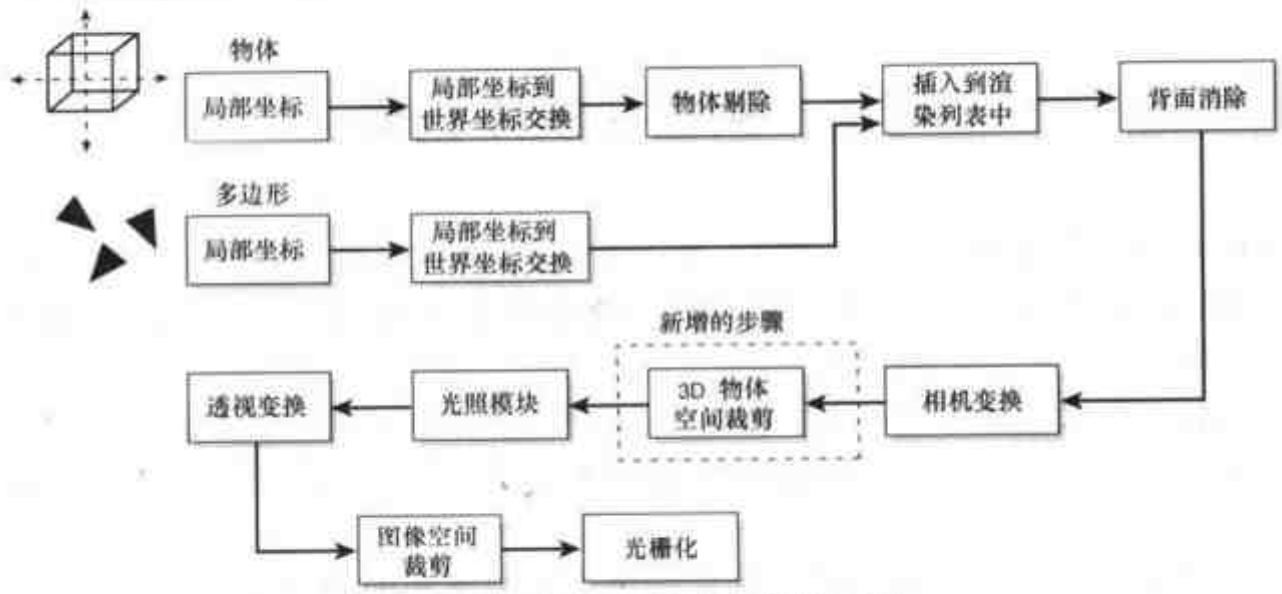


图 10.16　包含 3D 裁剪功能的流水线

10.3.1　几何流水线和数据结构

从图 10.16 可知，以完整物体（或独立多边形）的方式将几何体加载到系统中，对物体进行变换和剔除后，将其转换为多边形，再执行背面消除和另一种变换；然后执行裁剪、剔除和光照处理并在此进行变换，最后将其显示到屏幕上。执行的变换很多。关键之处在于，我们是以物体开始的，对于没有室内环境或室外环境的游戏来说，这很好；但对于有这些东西的游戏来说，这非常糟糕。

我们不能再局限于“物体”，需要创建更适合用于表示地形、室内环境等东西的实体。这也是我们进行光照计算和剔除操作时，很少使用基于物体的函数的原因所在。希望读者从多边形的角度考虑问题，稍后我们将创建一些更高级的数据结构，以支持大型物体。然而，我们不希望这些物体仅仅是一个大型多边形网格，同时希望它们是有序、分类、相关联、分层的。

例如，图 10.17 是一个我们可能要模拟的室外城市游戏（city game）关卡的俯视图。虽然可以将关卡数据存储在一个类似于物体的容器中，但我们希望有另一种数据结构，让我们能够快速剔除几何体的子部件。这是 BSP（二元空间划分）树、入口、四叉树等概念发挥作用的地方。

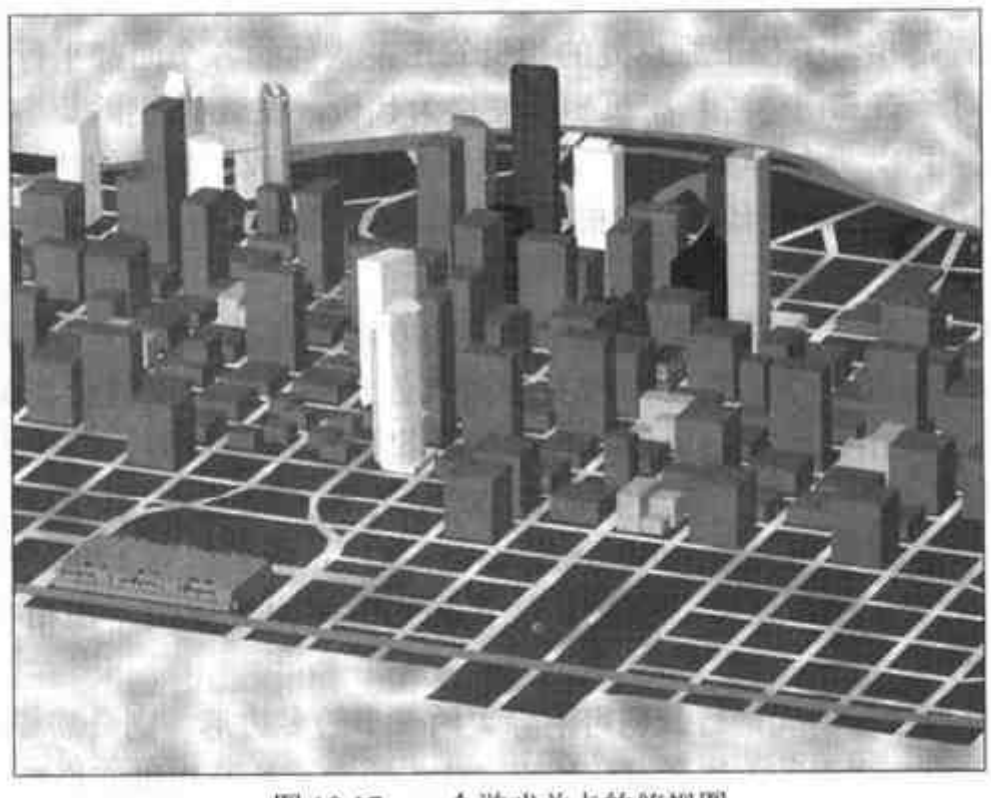


图 10.17　一个游戏关卡的俯视图

要使用这些概念，必须能够高效地实现它们，并在某个地方将其转换为多边形。作者认为，必须考虑要创建的游戏类型，并认识到在各种室内/室外游戏中，裁剪（包括在多边形剔除）至关重要，因此最好能够对多边形进行裁剪和剔除。我们之所以在渲染列表而不是物体中添加对多边形裁剪和剔除的支持，是因为我们不能再假定什么东西都可以用物体数据结构表示了。

10.3.2　在引擎中加入裁剪功能

包含裁剪代码的库模块名为 T3DLIB8.CPP|H，它很小，只有几个函数。和往常一样，这里只概述库模块，而不讨论为何需要这些函数。下面介绍头文件的各个部分。

1. 头文件

为裁剪函数定义的常量不多，它们用于控制裁剪函数的逻辑以及根据哪些裁剪面进行裁剪。这些常量如下：

```
// 多边形裁剪标记
#define CLIP_POLY_X_PLANE          0x0001 // 根据左、右裁剪面进行裁剪
#define CLIP_POLY_Y_PLANE          0x0002 // 根据上、下裁剪面进行裁剪
#define CLIP_POLY_Z_PLANE          0x0004 // 根据远、近裁剪面进行裁剪
#define CLIP_OBJECT_XYZ_PLANES  (CULL_OBJECT_X_PLANE | CULL_OBJECT_Y_PLANE |
        CULL_OBJECT_Z_PLANE)
```

这个库模块中只有两个函数，一个用于裁剪，另一个用于生成地形：

```
void Clip_Polys_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                  //要裁剪的渲染列表
CAM4DV1_PTR cam,//相机
int clip_flags);//裁剪标记，要根据哪些裁剪面进行裁剪

int Generate_Terrain_OBJECT4DV2(OBJECT4DV2_PTR obj,   // 指向物体的指针
 float twidth,       // 在世界坐标系中的宽度（X 轴）
 float theight,      // 在世界坐标系中的长度（z 轴）
 float vscale,       // 地形中最大可能高度
 char *height_map_file, // 256 色高位图的文件名
 char *texture_map_file, // 纹理图的文件名
 int rgbcolor,       // 没有纹理时地形的颜色
 VECTOR4D_PTR pos,    // 初始位置
 VECTOR4D_PTR rot,    // 初始旋转角度
 int poly_attr);     // 着色属性
```

接下来讨论裁剪函数的工作原理。

2. 创建裁剪函数

该裁剪函数在相机坐标空间中对渲染列表中的多边形进行裁剪，这意味着相机位于原点，并根据+z 轴调整了裁剪面的位置（如图 10.18 所示）。

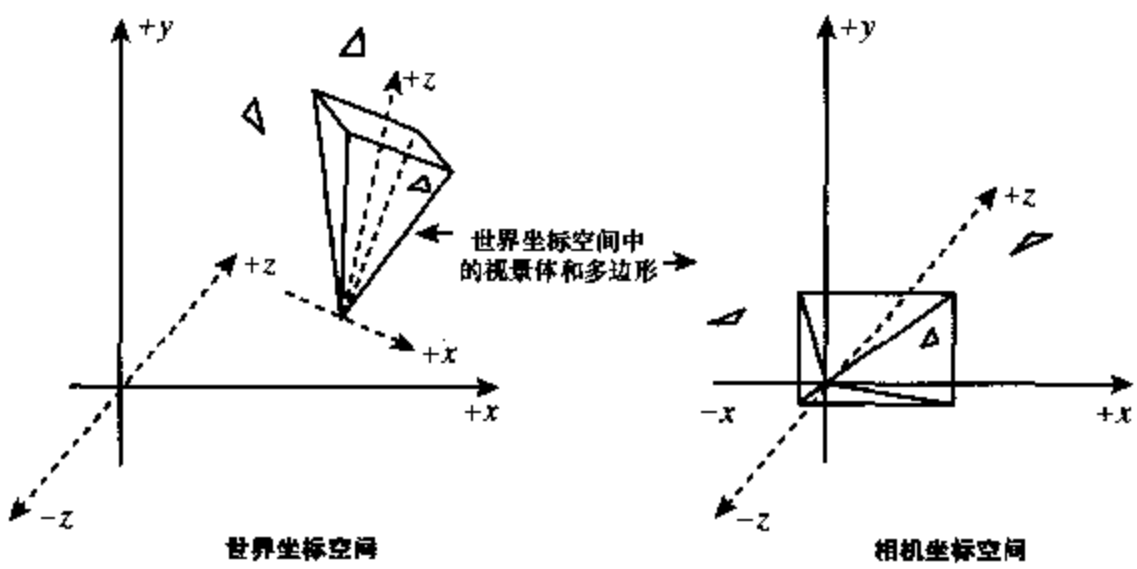


图 10.18　相机坐标空间中的裁剪

下面是在当前的流水线中渲染 3D 图像的函数调用流程：

```
Reset_OBJECT4DV2(&obj);

// 创建一个单位矩阵
MAT_IDENTITY_4X4(&mtrans);

// 对物体进行变换
```

```
Transform_OBJECT4DV2(&obj, &mtrans, TRANSFORM_LOCAL_TO_TRANS,1);

// 局部坐标到世界坐标变换
Model_To_World_OBJECT4DV2(&obj, TRANSFORM_TRANS_ONLY);

// 将物体插入到渲染列表中
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &obj,0);

// 背面消除
Remove_Backfaces_RENDERLIST4DV2(&rend_list, &cam);

// 世界坐标到相机坐标变换
World_To_Camera_RENDERLIST4DV2(&rend_list, &cam);

//////////////////////////////////////////////////////////
// 在这里使用视景体对多边形进行裁剪
//////////////////////////////////////////////////////////

// 一次性对场景执行光照计算
Light_RENDERLIST4DV2_World16(&rend_list, &cam, lights, 4);

// 对多边形列表进行排序
Sort_RENDERLIST4DV2(&rend_list, SORT_POLYLIST_AVGZ);

// 相机坐标到透视坐标变换
Camera_To_Perspective_RENDERLIST4DV2(&rend_list, &cam);

// 屏幕变换
Perspective_To_Screen_RENDERLIST4DV2(&rend_list, &cam);

// 锁定后缓存
DDraw_Lock_Back_Surface();

// 渲染多边形
Draw_RENDERLIST4DV2_Solid16(&rend_list, back_buffer, back_lpitch);

// 解除对后缓存的锁定
DDraw_Unlock_Back_Surface();
```

作者使用粗体文本指出了要在流水线的什么地方对多边形进行裁剪：执行相机变换之后，但在执行光照计算之前。这很好，通过裁剪，不但可以减少多边形数量，还可避免对被剔除的多边形执行光照计算。

下面讨论为对渲染列表执行裁剪，需要完成哪些工作。渲染列表由两个数据结构组成：一个多边形数组，一个多边形指针数组。使用多边形指针数组旨在提高排序速度，这样无需重新分配和释放存储空间——存储多边形的空间始终保持不变。

（1）根据上/下或左/右裁剪面进行剔除

我们不根据视景体的上/下或左/右裁剪面对多边形进行裁剪，而只是检测多边形是否完全在这些裁剪面的外侧或内侧。如果多边形完全在裁剪面的内侧，则保留不变，并进入下一个多边形；如果完全在外侧，则将其剔除（保留或丢弃裁剪，no-holds-barred clipping），如图 10.19 所示。

为此，只需要知道检测的三角形的顶点以及裁剪面的位置和朝向。如果知道裁剪面的点法线格式平面方程，可以通过计算法线和平面上的点到顶点的向量的点积，来确定顶点位于哪个半空间中。如果全部三个顶点都位于裁剪面的同一侧，便可以进行简单的拒绝/接受。

我们知道，裁剪面都与相机坐标轴平行或垂直，因此可以根据裁剪面的斜率或视野角来判断顶点是否在视景体内。在如图 10.19 中，根据视景体的右裁剪面来检测多边形 1。我们可以暂时不管 y 坐标，而是考虑 xz 平面。问题是：顶点 $v1(x1, y1, z1)$ 在右裁剪面内侧还是外侧？

在图 10.19 中，视野角为 90 度，这意味着左、右裁剪面与 z 轴之间的夹角都是 45 度，因此右裁剪面的平面方程为 $x=z$。这意味着如果 $x1=z1$，则顶点 $v1$ 位于裁剪面上；如果 $x1>z1$，则位于外侧；如果 $x1<z1$，则位于内侧。

一般而言，视野角为 90 度时，给定顶点 $v(x, y, z)$：

- 如果 $x>z$，则 v 位于右裁剪面外侧；

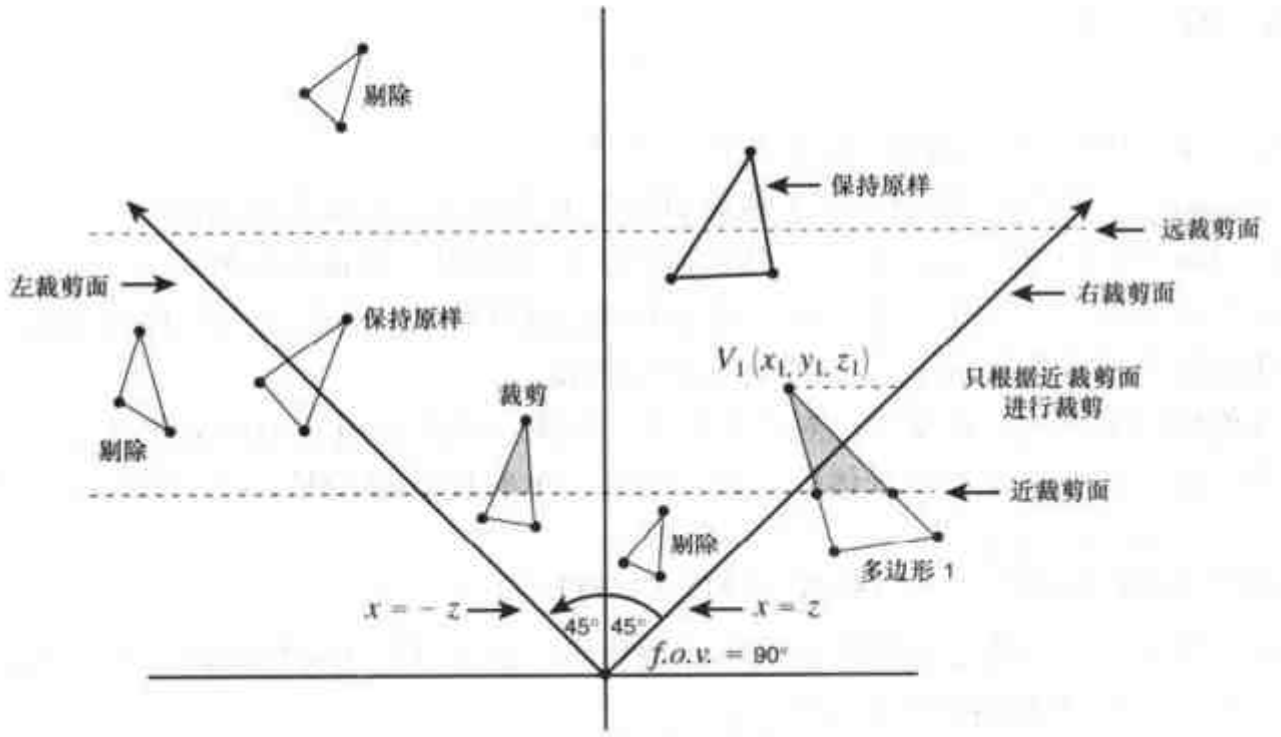


图 10.19　保留或丢弃裁剪

- 如果 $x<z$，则 v 位于右裁剪面内侧；
- 如果 $x=z$，则 v 位于右裁剪面上。

同样，对于左侧裁剪面：

- 如果 $x>-z$，则 v 位于左裁剪面内侧；
- 如果 $x<-z$，则 v 位于左裁剪面外侧；
- 如果 $x=-z$，则 v 位于左裁剪面上。

上/下裁剪面的情况与此相同，只需将 x 替换为 y 即可。当然，这是一个特例，只适用于视野角为 90 度的情况；但改变视野角只会改变裁剪面的斜率，因此只需稍微修改上述公式，以考虑到视野角。图 10.20 说明了左右裁剪面的一般性情形：

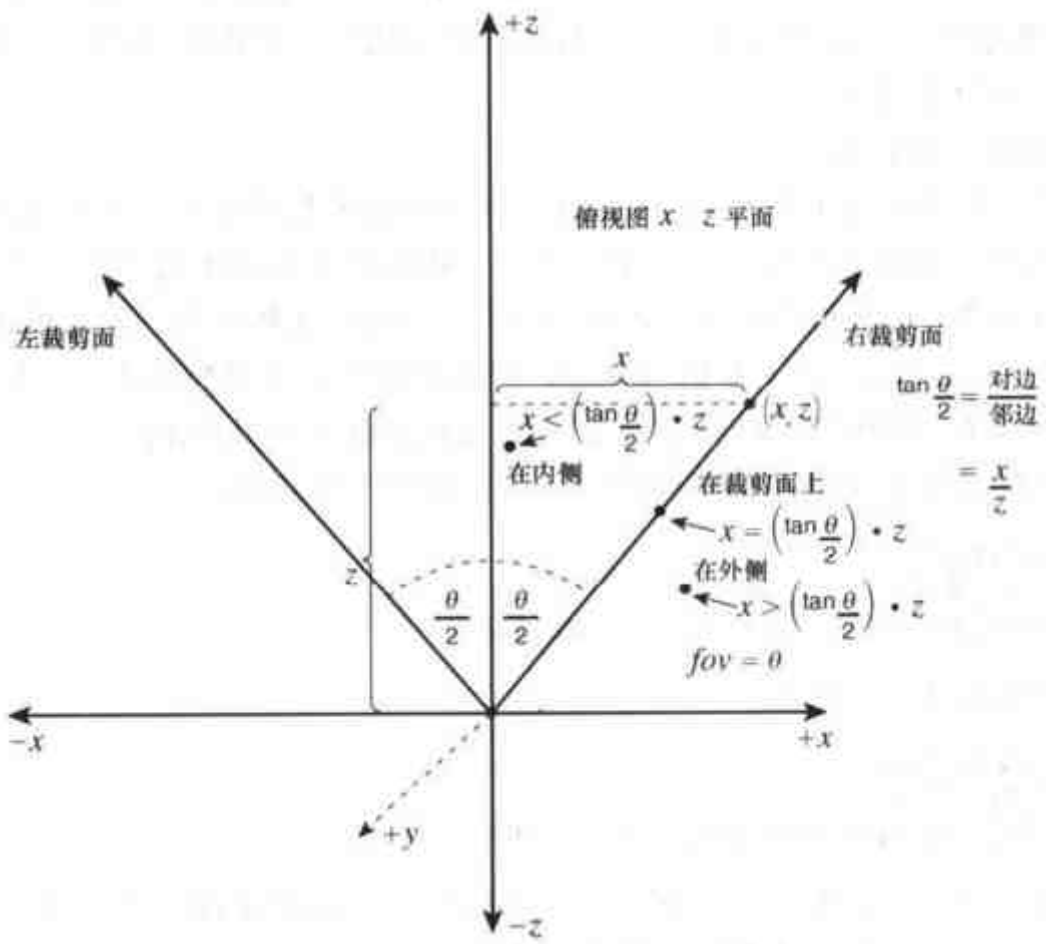


图 10.20　根据裁剪面的斜率进行剔除检测

$\tan(\theta/2) = |x|/z$

因此：

$|x| = \tan(\theta/2) * z$

换句话说：

- 如果$|x| = \tan(\theta/2) * z$，则顶点位于裁剪面上；
- 如果$|x| < \tan(\theta/2) * z$，则顶点位于裁剪面的正半空间内，即裁剪面内侧；
- 如果$|x| < \tan(\theta/2) * z$，则顶点位于裁剪面的负半空间内，即裁剪面外侧。

需要根据 4 个裁剪面（上、下、左、右）对每个顶点进行测试，看它是否在裁剪面的内侧。这意味着需要根据视野角将每个顶点的坐标 x、y 同坐标 z 进行比较。

这种比较我们做过很多次，读者应该对此很熟悉。如果三角形的所有顶点都在视景体内，则跳到渲染列表中的下一个三角形；如果都在视景体外，则将整个三角形裁剪/剔除掉，并将其标记为“被裁剪掉”：

```
// 裁剪掉完全在视景体外的多边形
SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);
```

为节省篇幅，作者将在讨论 z 裁剪时再列出函数代码。这里只列出实现前面介绍的测试的代码，下面是检测单个顶点是否在左/右裁剪面内侧的代码：

```
// 由于需要根据左/右裁剪面进行裁剪
// 因此需要根据 FOV 或裁剪面方程来确定在当前 x 值的情况下，
//z 值为多少时点将位于裁剪面外侧
z_factor = (0.5)*cam->viewplane_width/cam->view_dist;

// 顶点 0
z_test = z_factor*curr_poly->tvlist[0].z;
if (curr_poly->tvlist[0].x > z_test)
  vertex_ccodes[0] = CLIP_CODE_GX;
else
if (curr_poly->tvlist[0].x < -z_test)
  vertex_ccodes[0] = CLIP_CODE_LX;
else
  vertex_ccodes[0] = CLIP_CODE_IX;
```

我们只需知道顶点在裁剪面的哪一侧，然后据此设置每个顶点的裁剪码标记（类似于 Cohen-Sutherland 算法），以跟踪顶点在视景体内还是视景体外。这有助于后面执行简单拒绝/接受。下面介绍根据远/近裁剪面进行裁剪的算法，它要复杂得多。

（2）根据远/近裁剪面进行裁剪

这种裁剪有些棘手。裁剪说起来容易，但在 3D 空间实现起来却非常难。并不是算法或数学计算很难，而是其中的细节非常麻烦。剔除了所有在上、下、左、右裁剪面之外的多边形后，还必须根据远、近裁剪面对余下的多边形进行检测。这种检测很容易，因为这些裁剪面与 x 轴平行，而相机指向+z 轴方向。因此，对于三角形的每个顶点 $vi(xi, yi, zi)$（i 为 0、1、2），只需根据远、近裁剪面的 z 坐标对坐标 zi 进行测试，远、近裁剪面的 z 坐标存储在相机数据结构中，分别是 near_clip_z 和 far_clip_z。

例如，要检测多边形是否完全在远裁剪面外侧，可以使用下述算法：

```
如果(v0.z > far_clip_z)且
   (v1.z > far_clip_z)且
   (v2.z > far_clip_z) 则将多边形(v0, v1, v2)剔除
```

对于近裁剪面，测试算法与此类似：

```
if (v0.z < near_clip_z)且
  (v1.z < near_clip_z)且
  (v2.z < near_clip_z) 则将多边形(v0, v1, v2)剔除
```

我们并不关心三角形穿过远裁剪面，一部分在视景体内，一部分在视景体外的情况；但确实关心三角形穿过近裁剪面的情形。这个问题避不开的，解决方案如下：

```
for(渲染列表中的每个多边形 P)
begin
for(当前多边形 P 的每个顶点 v)
  begin
```

```
计算顶点 v 的位置：
1. v 位于视景体内？
2. v 位于远裁剪面的外侧？
3. v 位于近裁剪面的外侧？
end

if(全部 3 个顶点都在近裁剪面或远裁剪面外侧)then(将多边形 P 剔除)

if(如果至少有一个顶点位于视景体内且至少有一个顶点位于近裁剪面外侧)
   then(根据近裁剪面对多边形 P 进行裁剪)

end
```

其中以粗体显示的伪代码是接下来要实现的。我们需要拿出根据近裁剪面对多边形进行裁剪的方法，这意味着必须对穿过近裁剪面进入或离开视景体的每条边进行裁剪。这存在两种情形，如图 10.21 所示。

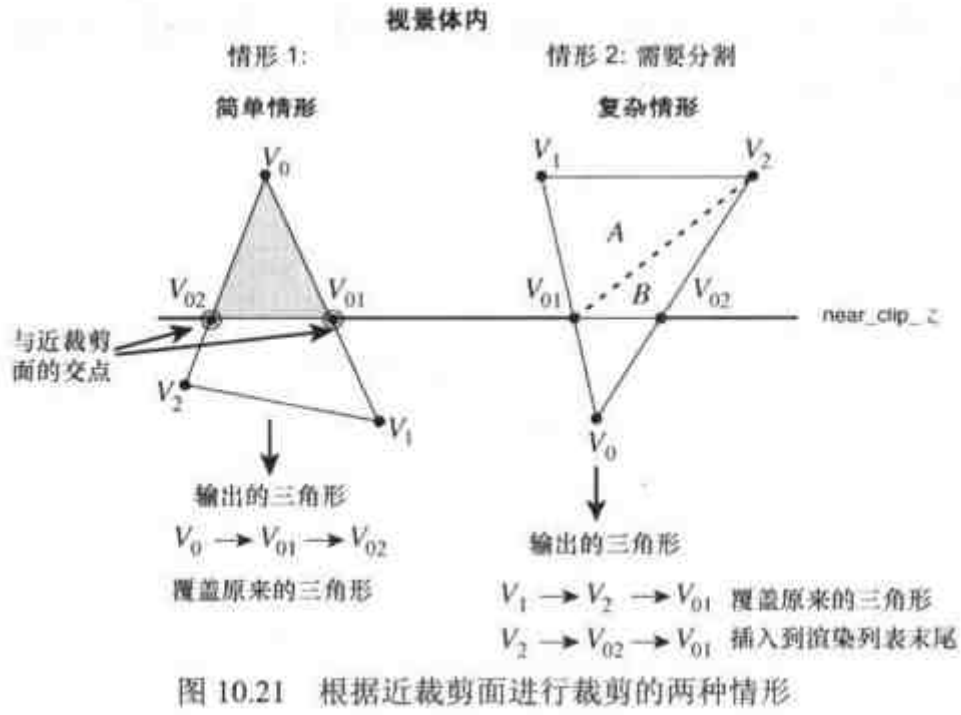


图 10.21　根据近裁剪面进行裁剪的两种情形

情形 1：一个顶点在视景体内，两个顶点在视景体外

这种情形很简单，只需对两条与近裁剪面相交的边进行裁剪，用新顶点覆盖原来的顶点即可。另外，如果三角形带纹理，还需要重新计算纹理坐标。当然，还必须重新计算法线，因为三角形缩小后，法线的长度也将发生变化。为提高光照计算的速度，我们将存储法线的长度，由于其长度发生了变化，因此必须重新计算。

情形 2：两个顶点在视景体内，一个顶点在视景体外

这种情形较复杂。如图 10.21 所示，当三角形有两个顶点位于视景体内，一个顶点位于视景体外时，三角形与近裁剪面之间有两个交点。这意味着裁剪后，三角形变成了四边形，而我们的引擎不能处理四边形，因此必须将其分割。我们需要采取的策略是，将原来的三角形的所有信息复制到一个临时三角形中，然后执行裁剪，并将裁剪得到的四边形分割成两个三角形：A 和 B。然后用三角形 A 覆盖原来的三角形；同时用交点数据更新临时三角形 B。这样，在渲染列表中，原来的三角形将被分割得到的三角形 A 代替，分割得到的另一个三角形 B 必须插入到渲染列表的末尾。

阅读后面的内容之前，读者一定要理解情形 2。简单地说，使用一个平面对三角形进行裁剪时，将得到一个或两个三角形。如果结果为一个三角形，可以用它覆盖原来的三角形，引起的变化不大。如果裁剪后得到两个三角形（A 和 B），则可以使用三角形 A 覆盖原来的三角形，但新增的三角形 B 必须插入到渲染列表末尾。

提示：读者可能认为，使用多边形指针数组而不是多边形链表实在太糟糕。原因在于，由于分割而新增的三角形被插入到渲染列表末尾，但其 z 值与原来的三角形相同，当对多边形进行排序时，必须将其移到渲染列表的前面，这将降低排序的速度。后面将对这种情况进行优化。

最后需要解决的细节是裁剪代码。作者决定组合前面讨论过的各种裁剪算法，编写一种混合算法。我们已经完成了预处理步骤：设置顶点的裁剪码标记；余下的任务是如何计算三角形边与近裁剪面的交点。有两种情形需要处理（一个顶点位于裁剪区域内和两个顶点位于裁剪区域内），作者在代码中分别对其进行

处理。在每种情形下，都创建两条与近裁剪面相交的边的参数化线段方程，它们类似于这样：

```
p = v0 + (v1 - v0)*t
```

其中 v0 为起点，v1 为终点。

然后将 near_clip_z 代入到线段的 z 坐标参数化方程中，并求解 t：

```
near_clip_z = v0.z + (v1.z - v0.z)*t

t = near_clip_z - v0.z/(v1.z - v0.z)
```

然后将 t 值代入线段的 x、y 坐标参数化方程中，得到交点的 x、y 坐标。对两条边都执行这种操作后，任务就完成了。余下的只是一个清理工作：使用正确的顶点数据和交点覆盖原来的三角形。

（3）计算纹理坐标

余下的一个细节是计算纹理坐标。如果多边形有纹理坐标，也必须对其进行裁剪，幸运的是，可以使用前述 t 值来计算裁剪后的纹理坐标。下面举一个例子。假设有一个属于情形 1 的三角形，如图 10.22 所示。该三角形的每个顶点都有纹理坐标，如图中所示。

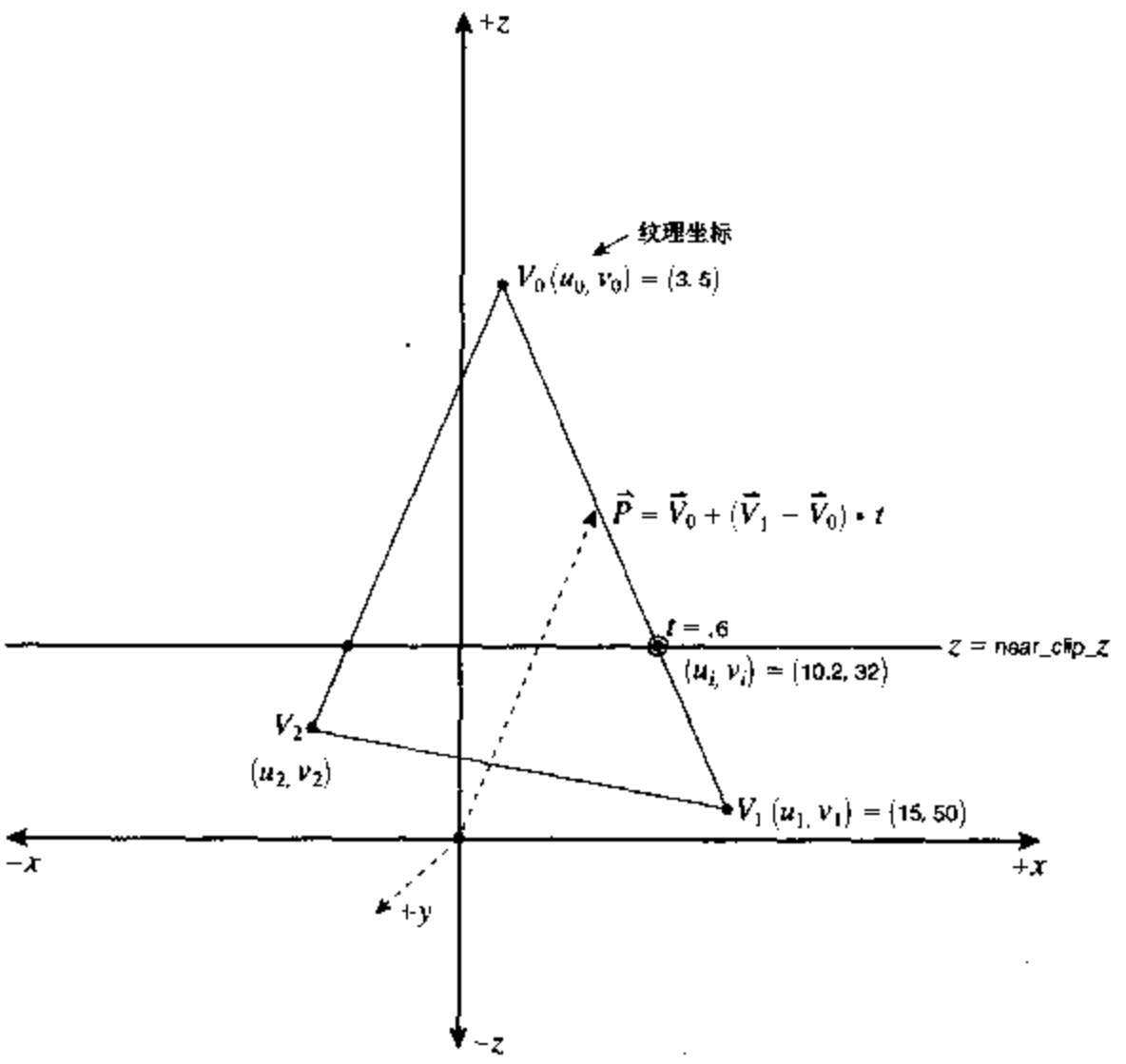


图 10.22　计算裁剪后的三角形的纹理坐标

现在假设我们已经计算出交点，交点的 t 值为 0.6。只需使用它对纹理坐标进行线性插值，计算出裁剪后的纹理坐标。请看图 10.22，顶点 0 的纹理坐标为(3，5)，顶点 1 的纹理坐标为(15，50)。计算裁剪后的纹理坐标的公式如下：

```
u_clipped = u0 + (u1 - u0)*t
v_clipped = v0 + (v1 - v0)*t
```

将纹理坐标代入上述公式，结果如下：

```
u_clipped = 3 + (15 - 3)*(.6) = 10.2
v_clipped = 5 + (50 - 5)*(.6) = 32
```

最后，还需要在两种情形下重新计算多边形法线的长度，因为在光照阶段需要使用法线长度，而它们已经发生了变化。为此，只需对裁剪（并分割）后的每个三角形执行下述代码：

```
// 计算 u 和 v
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v1].v, &u);
VECTOR4D_Build(&curr_poly->tvlist[v0].v, &curr_poly->tvlist[v2].v, &v);

// 计算叉积
VECTOR4D_Cross(&u, &v, &n);

// 计算并存储法线长度
curr_poly->nlength = VECTOR4D_Length_Fast(&n);
```

这是一项棘手的任务，但在每一帧中，需要使用近裁剪面对其进行裁剪的多边形通常只有几百个，作者使用函数 VECTOR4D_Length_Fast 来计算法线长度，它比较粗糙，但速度比计算平方根快得多。

至此，对裁剪的介绍就结束了，来看一看裁剪函数的代码：

```
void Clip_Polys_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
                CAM4DV1_PTR cam, int clip_flags)
{
// 这个函数使用指定的裁剪面对渲染列表中的多边形进行裁剪
// 并设置多边形的裁剪标记，以免渲染被裁剪掉的多边形
// 该函数只使用远、近裁剪面进行裁剪
// 但将根据上/下、左/右裁剪面进行简单拒绝/接受测试
// 如果多边形完全在视景体外，将被剔除掉
// 这种测试的效率没有基于物体的测试高，因为可见的物体包含可见的多边形
// 但当多边形列表是基于总有一部分可见的大型物体时，对各个多边形进行检测是值得的
// 这个函数假设已经对多边形执行相机变换

// 内部裁剪码
#define CLIP_CODE_GZ   0x0001    // z > z_max
#define CLIP_CODE_LZ   0x0002    // z < z_min
#define CLIP_CODE_IZ   0x0004    // z_min < z < z_max

#define CLIP_CODE_GX   0x0001    // x > x_max
#define CLIP_CODE_LX   0x0002    // x < x_min
#define CLIP_CODE_IX   0x0004    // x_min < x < x_max

#define CLIP_CODE_GY   0x0001    // y > y_max
#define CLIP_CODE_LY   0x0002    // y < y_min
#define CLIP_CODE_IY   0x0004    // y_min < y < y_max

#define CLIP_CODE_NULL 0x0000

int vertex_ccodes[3]; // 用于存储裁剪标记
int num_verts_in;     // 位于视景体内部的顶点数
int v0, v1, v2;       // 顶点索引

float z_factor,       // 用于裁剪计算
   z_test;         // 用于裁剪计算

float xi, yi, x01i, y01i, x02i, y02i, // 交点坐标
   t1, t2,              // 参数化 t 值
   ui, vi, u01i, v01i, u02i, v02i; // 交点纹理坐标

int last_poly_index,   // 渲染列表中最后一个有效的多边形
  insert_poly_index; // 新多边形的插入位置

VECTOR4D u,v,n;      // 用于向量计算

POLYF4DV2 temp_poly; // 将多边形分割成两个时，用于存储新增的多边形

// 设置 last_poly_index 和 insert_poly_index，使其对应于渲染列表末尾
// 以免分割多边形两次
insert_poly_index = last_poly_index = rend_list->num_polys;

// 遍历渲染列表，对其中的多边形进行裁剪/剔除
for (int poly = 0; poly < last_poly_index; poly++)
 {
```

```
// 获得当前多边形
POLYF4DV2_PTR curr_poly = rend_list->poly_ptrs[poly];

// 该多边形有效吗?
// 仅当多边形处于活动状态、为被裁剪掉且不是背面时才对其进行处理
if ((curr_poly==NULL) || !(curr_poly->state & POLY4DV2_STATE_ACTIVE) ||
  (curr_poly->state & POLY4DV2_STATE_CLIPPED ) ||
  (curr_poly->state & POLY4DV2_STATE_BACKFACE) )
  continue; // 进入下一个多边形

  // 根据左、右裁剪面进行裁剪/剔除
  if (clip_flags & CLIP_POLY_X_PLANE)
    {
    // 只根据左、右裁剪面进行裁剪/剔除
    // 对于每个顶点，判断它是否在裁剪区域内
   // 并据此设置相应的裁剪码
   // 不对三角形进行裁剪，只进行简单拒绝测试
   // 将在光栅化函数中根据屏幕对多边形进行裁剪
   // 但要剔除完全在视景体外的多边形

   // 由于是根据左/右裁剪面进行裁剪
   // 因此需要根据 FOV 或裁剪面方程来确定，
   // x 和 z 满足什么条件时点将在裁剪面的外侧
   z_factor = (0.5)*cam->viewplane_width/cam->view_dist;

   // 顶点 0

   z_test = z_factor*curr_poly->tvlist[0].z;

   if (curr_poly->tvlist[0].x > z_test)
    vertex_ccodes[0] = CLIP_CODE_GX;
   else
   if (curr_poly->tvlist[0].x < -z_test)
    vertex_ccodes[0] = CLIP_CODE_LX;
   else
    vertex_ccodes[0] = CLIP_CODE_IX;

   // 顶点 1

   z_test = z_factor*curr_poly->tvlist[1].z;

   if (curr_poly->tvlist[1].x > z_test)
    vertex_ccodes[1] = CLIP_CODE_GX;
   else
   if (curr_poly->tvlist[1].x < -z_test)
    vertex_ccodes[1] = CLIP_CODE_LX;
   else
    vertex_ccodes[1] = CLIP_CODE_IX;

   // 顶点 2

   z_test = z_factor*curr_poly->tvlist[2].z;

   if (curr_poly->tvlist[2].x > z_test)
    vertex_ccodes[2] = CLIP_CODE_GX;
   else
   if (curr_poly->tvlist[2].x < -z_test)
    vertex_ccodes[2] = CLIP_CODE_LX;
   else
    vertex_ccodes[2] = CLIP_CODE_IX;

   // 进行简单拒绝测试，即多边形是否完全在左裁剪面或右裁剪面外侧
   if ( ((vertex_ccodes[0] == CLIP_CODE_GX) &&
      (vertex_ccodes[1] == CLIP_CODE_GX) &&
      (vertex_ccodes[2] == CLIP_CODE_GX) ) ||

      ((vertex_ccodes[0] == CLIP_CODE_LX) &&
```

```
      (vertex_ccodes[1] == CLIP_CODE_LX) &&
      (vertex_ccodes[2] == CLIP_CODE_LX) ) )

    {
    // 将完全在裁剪面外侧的多边形裁剪掉
    SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

    // 进入下一个多边形
    continue;
    } // end if

 } // end if x planes

// 根据上/下裁剪面进行裁剪/剔除
if (clip_flags & CLIP_POLY_Y_PLANE)
  {
  // 只根据上/下裁剪面进行裁剪/剔除
  // 对于每个顶点，判断它是否在裁剪区域内
  // 并据此设置相应的裁剪码
  // 不对三角形进行裁剪，只进行简单拒绝测试
  // 将在光栅化函数中根据屏幕对多边形进行裁剪
  // 但要剔除完全在视景体外的多边形

  // 由于是根据上/下裁剪面进行裁剪
  // 因此需要根据 FOV 或裁剪面方程来确定，
  // y 和 z 满足什么条件时点将在裁剪面的外侧
  z_factor = (0.5)*cam->viewplane_width/cam->view_dist;

  // 顶点 0

  z_test = z_factor*curr_poly->tvlist[0].z;

  if (curr_poly->tvlist[0].y > z_test)
   vertex_ccodes[0] = CLIP_CODE_GY;
  else
  if (curr_poly->tvlist[0].y < -z_test)
   vertex_ccodes[0] = CLIP_CODE_LY;
  else
   vertex_ccodes[0] = CLIP_CODE_IY;

  // 顶点 1

  z_test = z_factor*curr_poly->tvlist[1].z;

  if (curr_poly->tvlist[1].y > z_test)
   vertex_ccodes[1] = CLIP_CODE_GY;
  else
  if (curr_poly->tvlist[1].y < -z_test)
   vertex_ccodes[1] = CLIP_CODE_LY;
  else
   vertex_ccodes[1] = CLIP_CODE_IY;

  // 顶点 2

  z_test = z_factor*curr_poly->tvlist[2].z;

  if (curr_poly->tvlist[2].y > z_test)
   vertex_ccodes[2] = CLIP_CODE_GY;
  else
  if (curr_poly->tvlist[2].x < -z_test)
   vertex_ccodes[2] = CLIP_CODE_LY;
  else
   vertex_ccodes[2] = CLIP_CODE_IY;

  // 进行简单拒绝测试，即多边形是否完全在上裁剪面或下裁剪面外侧
  if ( ((vertex_ccodes[0] == CLIP_CODE_GY) &&
     (vertex_ccodes[1] == CLIP_CODE_GY) &&
```

```
     (vertex_ccodes[2] == CLIP_CODE_GY) ) ||

     ((vertex_ccodes[0] == CLIP_CODE_LY) &&
     (vertex_ccodes[1] == CLIP_CODE_LY) &&
     (vertex_ccodes[2] == CLIP_CODE_LY) ) )

    {
    // 将完全在裁剪面外侧的多边形裁剪掉
    SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

    // 进入下一个多边形
    continue;
    } // end if

  } // end if y planes

  // 根据远/近裁剪面进行裁剪/剔除
  if (clip_flags & CLIP_POLY_Z_PLANE)
    {
    // 只根据远/近裁剪面进行裁剪/剔除
    // 对于每个顶点，判断它是否在裁剪区域内
    // 并据此设置相应的裁剪码
    // 然后根据近裁剪面对多边形进行裁剪
    // 这最多会增加一个三角形

    // 重置内部顶点计数器，该变量用于对最终的三角形进行分类
    num_verts_in = 0;

    // 顶点 0
    if (curr_poly->tvlist[0].z > cam->far_clip_z)
     {
     vertex_ccodes[0] = CLIP_CODE_GZ;
     }
    else
     if (curr_poly->tvlist[0].z < cam->near_clip_z)
     {
     vertex_ccodes[0] = CLIP_CODE_LZ;
     }
    else
     {
     vertex_ccodes[0] = CLIP_CODE_IZ;
     num_verts_in++;
     }

    // 顶点 1
    if (curr_poly->tvlist[1].z > cam->far_clip_z)
     {
     vertex_ccodes[1] = CLIP_CODE_GZ;
     }
    else
    if (curr_poly->tvlist[1].z < cam->near_clip_z)
     {
     vertex_ccodes[1] = CLIP_CODE_LZ;
     }
    else
     {
     vertex_ccodes[1] = CLIP_CODE_IZ;
     num_verts_in++;
     }

    // 顶点 2
    if (curr_poly->tvlist[2].z > cam->far_clip_z)
     {
     vertex_ccodes[2] = CLIP_CODE_GZ;
     }
    else
    if (curr_poly->tvlist[2].z < cam->near_clip_z)
```

```
    {
    vertex_ccodes[2] = CLIP_CODE_LZ;
    }
   else
    {
    vertex_ccodes[2] = CLIP_CODE_IZ;
    num_verts_in++;
    }

  // 进行简单拒绝测试，即多边形是否完全在远裁剪面或近裁剪面外侧
  if ( ((vertex_ccodes[0] == CLIP_CODE_GZ) &&
    (vertex_ccodes[1] == CLIP_CODE_GZ) &&
    (vertex_ccodes[2] == CLIP_CODE_GZ) ) ||

    ((vertex_ccodes[0] == CLIP_CODE_LZ) &&
    (vertex_ccodes[1] == CLIP_CODE_LZ) &&
    (vertex_ccodes[2] == CLIP_CODE_LZ) ) )

  {
  // 将完全在裁剪面外侧的多边形裁剪掉
  SET_BIT(curr_poly->state, POLY4DV2_STATE_CLIPPED);

  // 进入下一个多边形
  continue;
  } // end if

// 判断是否有顶点在近裁剪面外侧
if ( ( (vertex_ccodes[0] | vertex_ccodes[1] |
     vertex_ccodes[2]) & CLIP_CODE_LZ) )
{
// 至此可以根据近裁剪面进行裁剪了
// 不需要根据远裁剪面进行裁剪，因为即使多边形穿过远裁剪面也不会引发问题
// 裁剪时有两种情形
// 情形 1：三角形有 1 个顶点在近裁剪面内侧，2 个顶点在外侧
// 情形 2：三角形有 2 个顶点在近裁剪面内侧，1 个顶点在外侧

// 第 1 步：根据内外侧顶点数对三角形进行分类
// 情形 1：简单情形
if (num_verts_in == 1)
  {
  // 需要根据近裁剪面对三角形进行裁剪
  // 对包含内侧顶点的每条边进行裁剪
  // 为此，需要根据边的参数化方程计算它与近裁剪面的交点
  // 然后用交点替换原来位于外侧的三角形顶点，并重新计算纹理坐标（如果有的话）
  // 在这种情形下，不会增加三角形，因此只需用裁剪后的三角形覆盖原来的三角形
  // 在另一种情形下，将得到两个三角形，
  // 因此至少需要将一个三角形插入到渲染列表末尾

  // 第 1 步：找出位于内侧的顶点
  if ( vertex_ccodes[0] == CLIP_CODE_IZ)
   { v0 = 0; v1 = 1; v2 = 2; }
  else
  if (vertex_ccodes[1] == CLIP_CODE_IZ)
   { v0 = 1; v1 = 2; v2 = 0; }
  else
   { v0 = 2; v1 = 0; v2 = 1; }

  // 第 2 步：对每条边进行裁剪
  // 创建参数化线段方程 p = v0 + v01*t
  // 将 near_clip_z 代入 z 坐标分量方程，以求出 t 值
  // 然后将 t 值代入 x、y 坐标分量方程，计算出交点的 x、y 坐标

  // 对三角形边 v0->v1 进行裁剪
  VECTOR4D_Build(&curr_poly->tvlist[v0].v,
         &curr_poly->tvlist[v1].v, &v);

  // 交点的 z 坐标为 near_clip_z，因此 t 值如下
```

```
t1 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

// 将 t 值代入 x、y 坐标分量方程，得到交点的 x、y 坐标
xi = curr_poly->tvlist[v0].x + v.x * t1;
yi = curr_poly->tvlist[v0].y + v.y * t1;

// 用交点覆盖原来的顶点
curr_poly->tvlist[v1].x = xi;
curr_poly->tvlist[v1].y = yi;
curr_poly->tvlist[v1].z = cam->near_clip_z;

// 对三角形边 v0->v2 进行裁剪
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
       &curr_poly->tvlist[v2].v, &v);

// 交点的 z 坐标为 near_clip_z，因此 t 值如下
t2 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

// 将 t 值代入 x、y 坐标分量方程，得到交点的 x、y 坐标
xi = curr_poly->tvlist[v0].x + v.x * t2;
yi = curr_poly->tvlist[v0].y + v.y * t2;

// 用交点覆盖原来的顶点
curr_poly->tvlist[v2].x = xi;
curr_poly->tvlist[v2].y = yi;
curr_poly->tvlist[v2].z = cam->near_clip_z;

// 检查多边形是否带纹理
// 如果是，则对纹理坐标进行裁剪
if (curr_poly->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
  {
  ui = curr_poly->tvlist[v0].u0 +
    (curr_poly->tvlist[v1].u0 - curr_poly->tvlist[v0].u0)*t1;
  vi = curr_poly->tvlist[v0].v0 +
    (curr_poly->tvlist[v1].v0 - curr_poly->tvlist[v0].v0)*t1;
  curr_poly->tvlist[v1].u0 = ui;
  curr_poly->tvlist[v1].v0 = vi;

  ui = curr_poly->tvlist[v0].u0 +
    (curr_poly->tvlist[v2].u0 - curr_poly->tvlist[v0].u0)*t2;
  vi = curr_poly->tvlist[v0].v0 +
    (curr_poly->tvlist[v2].v0 - curr_poly->tvlist[v0].v0)*t2;
  curr_poly->tvlist[v2].u0 = ui;
  curr_poly->tvlist[v2].v0 = vi;
  } // end if textured

  // 最后，需要重新计算法线长度

  // 计算 u 和 v
  VECTOR4D_Build(&curr_poly->tvlist[v0].v,
         &curr_poly->tvlist[v1].v, &u);
  VECTOR4D_Build(&curr_poly->tvlist[v0].v,
         &curr_poly->tvlist[v2].v, &v);

  // 计算叉积
  VECTOR4D_Cross(&u, &v, &n);

  // 计算并存储法线长度
  curr_poly->nlength = VECTOR4D_Length_Fast(&n);

  } // end if
else
if (num_verts_in == 2)
  {

  // 对包含外侧顶点的每条边进行裁剪
  // 为此，需要根据边的参数化方程计算它与近裁剪面的交点
```

```
// 然而，不同于情形 1，这里三角形被裁剪后为四边形，需要划分成两个三角形
// 因此我们用其中一个三角形覆盖原来的三角形，
// 将另一个三角形插入到渲染列表末尾

// 第 0 步：复制多边形
memcpy(&temp_poly, curr_poly, sizeof(POLYF4DV2) );

// 第 1 步：找出位于外侧的顶点
if ( vertex_ccodes[0] == CLIP_CODE_LZ)
 { v0 = 0; v1 = 1; v2 = 2; }
else
if (vertex_ccodes[1] == CLIP_CODE_LZ)
 { v0 = 1; v1 = 2; v2 = 0; }
else
 { v0 = 2; v1 = 0; v2 = 1; }

// 第 2 步：对每条边进行裁剪
// 创建参数化线段方程 p = v0 + v01*t
// 将 near_clip_z 代入 z 坐标分量方程，以求出 t 值
// 然后将 t 值代入 x、y 坐标分量方程，计算出交点的 x、y 坐标

// 对三角形边 v0->v1 进行裁剪
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
       &curr_poly->tvlist[v1].v, &v);

// 交点的 z 坐标为 near_clip_z，因此 t 值如下
t1 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

// 将 t 值代入 x、y 坐标分量方程，得到交点的 x、y 坐标
x01i = curr_poly->tvlist[v0].x + v.x * t1;
y01i = curr_poly->tvlist[v0].y + v.y * t1;

// 对三角形边 v0->v2 进行裁剪
VECTOR4D_Build(&curr_poly->tvlist[v0].v,
       &curr_poly->tvlist[v2].v, &v);

// 交点的 z 坐标为 near_clip_z，因此 t 值如下
t2 = ( (cam->near_clip_z - curr_poly->tvlist[v0].z) / v.z );

// 将 t 值代入 x、y 坐标分量方程，得到交点的 x、y 坐标
x02i = curr_poly->tvlist[v0].x + v.x * t2;
y02i = curr_poly->tvlist[v0].y + v.y * t2;

// 计算出交点后，需要用交点 1 覆盖原来三角形的顶点 0
//这是分割后的两个三角形中的第一个

// 用交点 1 覆盖原来三角形的顶点 0
curr_poly->tvlist[v0].x = x01i;
curr_poly->tvlist[v0].y = y01i;
curr_poly->tvlist[v0].z = cam->near_clip_z;

// 现在需要使用两个交点和原来的顶点 2 创建一个新的三角形
// 该三角形将被插入到渲染列表的末尾
// 但现在暂时将其存储在 temp_poly 中

// 因此保留 v2 不变，用 v01 覆盖 v1，v02 覆盖 v0
 temp_poly.tvlist[v1].x = x01i;
 temp_poly.tvlist[v1].y = y01i;
 temp_poly.tvlist[v1].z = cam->near_clip_z;

 temp_poly.tvlist[v0].x = x02i;
 temp_poly.tvlist[v0].y = y02i;
 temp_poly.tvlist[v0].z = cam->near_clip_z;

// 检查多边形是否带纹理
// 如果是，则对纹理坐标进行裁剪
if (curr_poly->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
```

```
{
// 计算多边形 1 的纹理坐标
u01i = curr_poly->tvlist[v0].u0 +
 (curr_poly->tvlist[v1].u0 - curr_poly->tvlist[v0].u0)*t1;
v01i = curr_poly->tvlist[v0].v0 +
 (curr_poly->tvlist[v1].v0 - curr_poly->tvlist[v0].v0)*t1;

 // 计算多边形 2 的纹理坐标
 u02i = curr_poly->tvlist[v0].u0 +
  (curr_poly->tvlist[v2].u0 - curr_poly->tvlist[v0].u0)*t2;
 v02i = curr_poly->tvlist[v0].v0 +
  (curr_poly->tvlist[v2].v0 - curr_poly->tvlist[v0].v0)*t2;

 // 覆盖原来的纹理坐标
 // 多边形 1
 curr_poly->tvlist[v0].u0 = u01i;
 curr_poly->tvlist[v0].v0 = v01i;

 // 多边形 2
 temp_poly.tvlist[v0].u0 = u02i;
 temp_poly.tvlist[v0].v0 = v02i;
 temp_poly.tvlist[v1].u0 = u01i;
 temp_poly.tvlist[v1].v0 = v01i;

 } // end if textured


 // 最后，需要重新计算法线长度

 // 首先计算多边形 1 的法线长度
 // 计算 u 和 v
 VECTOR4D_Build(&curr_poly->tvlist[v0].v,
        &curr_poly->tvlist[v1].v, &u);
 VECTOR4D_Build(&curr_poly->tvlist[v0].v,
        &curr_poly->tvlist[v2].v, &v);

 // 计算叉积
 VECTOR4D_Cross(&u, &v, &n);

 // 计算并存储法线长度
 curr_poly->nlength = VECTOR4D_Length_Fast(&n);

 // 现在计算多边形 2（temp_poly）的法线长度
 // build u, v
 VECTOR4D_Build(&temp_poly.tvlist[v0].v,
        &temp_poly.tvlist[v1].v, &u);
 VECTOR4D_Build(&temp_poly.tvlist[v0].v,
        &temp_poly.tvlist[v2].v, &v);

 // 计算叉积
 VECTOR4D_Cross(&u, &v, &n);

 // 计算并存储法线长度
 temp_poly.nlength = VECTOR4D_Length_Fast(&n);

 // 现在将多边形 2 插入到渲染列表末尾
 Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &temp_poly);

 } // end else

 } // end if near_z clipping has occured

 } // end if z planes

 } // end for poly

} // end Clip_Polys_RENDERLIST4DV2
```

作者很少阅读图书中列出的代码，原因是太过晦涩，看不懂。但有时候，作者也会逐行去阅读，尽力去理解代码的功能。作者喜欢抽象的解释和伪代码；但有些人的学习方式与此相反——这就是作者在本书中列出代码的原因。作者想说的是，请逐行阅读该函数的代码，并尽可能理解。它是很多工作的基础，实际上，该函数也很简单，其中很多代码是重复的。

这个函数的用法非常简单，可以像下面这样调用它：

```
Clip_Polys_RENDERLIST4DV2(&rend_list, &cam,
      CLIP_POLY_Z_PLANE | CLIP_POLY_X_PLANE | CLIP_POLY_Y_PLANE);
```

传递渲染列表、相机和控制标记后，其他的工作由函数去完成。可以独立地启用 x、y、z 裁剪，作者倾向于启用这三种裁剪，将那些会带来麻烦的多边形从流水线中剔除。

3. 裁剪演示程序

作为使用裁剪函数和阐述普遍原理的例子，作者编写了一个物体在空中旋转的演示程序。它包含在附带光盘中，名为 DEMOII10_1.CPP|EXE。和往常一样，要编译它，需要文件 T3DLIB1.CPP|H 到 T3DLIB8.CPP|H 以及所有的 DirectX .LIB 文件。图 10.23 是该演示程序的屏幕截图，它加载一个物体，将其显示到屏幕上，然后缓慢地旋转。用户可以移动物体，使之远离或靠近视平面，还可以旋转它以及使其移到屏幕之外。



图 10.23　裁剪演示程序的屏幕截图

这个演示程序的目的有两个。首先，进行物体剔除时，不能裁剪或剔除单个多边形，因此如果物体部分位于视景体内，将对整个物体进行光照计算和变换。现在情况不是这样了，因为我们对多边形进行裁剪/剔除。请注意屏幕底端显示的统计信息；屏幕上列出了控制演示程序运行的方法，总体而言，使用箭头键来移动物体，使用 O 键来更换物体，使用 X、Y、Z 键来分别启用/禁用 x、y、z 裁剪。

4. 更新光照系统

还有一个与光照系统相关的细节需要解决。在此之前，我们一直在世界坐标系中执行光照计算，光源和几何体的坐标都是世界坐标。然而，现在这样做将浪费大量的 CPU 周期。可以将光照计算推迟到执行裁剪后再进行，以避免对将被裁剪掉的多边形执行光照计算。为此，需要将光源变换为相机坐标（因为被裁剪后，几何体为相机坐标），并在相机坐标空间中执行光照计算。方法有两种。可以重新编写光照函数，根据相机（它是光照函数的一个参数）对光源的位置和方向进行变换。但这样做的问题是，如果基于物体来调用光照函数，将无缘无故地对光源的变换重复多次。一种更好的策略是，单独地将光源变换为相机坐标。

需要考虑的一个细节是，将光源位置和方向变换为相机坐标后，在什么地方存储原来的位置和方向？必须将光源数据结构从 LIGHTV1 升级为 LIGHTV2（2.0 版），并添加存储变换后的光源位置和方向的变量。另外，还需要重新编写每个涉及光源的函数，因为它们现在将使用结构 LIGHTV2 而不是 LIGHTV1。所幸的是，除了将字符串 LIGHTV1 替换为 LIGHTV2 外，几乎不需要做其他修改。来看一下新的 LIGHTV2 结构：

```
// 光源数据结构2.0版
typedef struct LIGHTV2_TYP
{
int state; // 光源状态
int id; // 光源 ID
int attr;  // 光源类型和其他属性

RGBAV1 c_ambient;   // 环境光强度
RGBAV1 c_diffuse;   // 散射光强度
RGBAV1 c_specular;  // 镜面反射光强度

POINT4D  pos;   // 光源位置（世界坐标系下和变化后）
POINT4D  tpos;
VECTOR4D dir;  // 光源方向（世界坐标系下和变化后）
VECTOR4D tdir;
float kc, kl, kq;   // 衰减因子
float spot_inner;   // 聚光灯内锥角
float spot_outer;   // 聚光灯外锥角
float pf;       // 聚光灯指数因子

int   iaux1, iaux2; // 辅助变量，供以后扩展时使用
float  faux1, faux2;
void *ptr;

} LIGHTV2, *LIGHTV2_PTR;
```

其中以粗体显示的是新增的变量，用于存储变换后的光源位置和方向。创建光源时，惟一不同的是，光源在世界坐标系中的位置和方向仍存储在 pos 和 dir 中，但它们的相机坐标版本存储在 tpos 和 tdir 中。问题是，如果对光源进行变换将如何呢？下面是执行这种功能的函数：

```
void Transform_LIGHTSV2(LIGHTV2_PTR lights,  // 要变换的光源数组
            int num_lights,    // 要变换的光源数
            MATRIX4X4_PTR mt,  // 变换矩阵
            int coord_select)  // 执行要对哪种坐标进行变换
{
// 这个函数使用指定的矩阵对所有光源进行变换
// 用于将光源放置到相机坐标空间中，
// 以确保在将多边形变换为相机坐标后调用光照函数时，光照计算是正确的
// 后面可能对该函数进行优化，通过判断光源类型来确定是否对其进行旋转
// 然而，场景中的顶点数以千计，少旋转几个点没有什么区别
// 注意：即使不需要对光源进行变换（即将在世界坐标空间中执行光照计算，也必须调用该函数）
// 在这种情况下，仍然必须将 pos 和 dir 的值复制到 tpos 和 tdir 中
// 即调用该函数时，将参数 coord_flag 设置为 TRANSFORM_COPY_LOCAL_TO_TRANS
// 将参数 mt 设置为 NULL

int curr_light; // 当前光源
MATRIX4X4 mr;   // 用于存储消除平移部分的矩阵

// 需要旋转光源的方向向量，但必须消除矩阵中的平移部分，否则结果将是错误的
// 因此复制变换矩阵，并将平移因子设置为 0

if (mt!=NULL)
  {
  MAT_COPY_4X4(mt, &mr);
  // 将平移因子设置为 0
  mr.M30 = mr.M31 = mr.M32 = 0;
  } // end if

// 要变换哪种坐标
switch(coord_select)
   {
   case TRANSFORM_COPY_LOCAL_TO_TRANS:
      {
      // 遍历所有光源
      for (curr_light = 0; curr_light < num_lights; curr_light++)
      {
```

```
    lights[curr_light].tpos = lights[curr_light].pos;
    lights[curr_light].tdir = lights[curr_light].dir;
    } // end for

    } break;

 case TRANSFORM_LOCAL_ONLY:
    {
    // 遍历所有光源
    for (curr_light = 0; curr_light < num_lights; curr_light++)
    {
    // 对局部/世界坐标进行变换，然后将变换结果存回去
    POINT4D presult; // 用于存储每次变换的结果

    // 对每个光源的位置进行变换
    Mat_Mul_VECTOR4D_4X4(&lights[curr_light].pos, mt, &presult);

    // 将结果存回去
    VECTOR4D_COPY(&lights[curr_light].pos, &presult);

    // 对方向向量进行变换
    Mat_Mul_VECTOR4D_4X4(&lights[curr_light].dir, &mr, &presult);

    // 将结果存回去
    VECTOR4D_COPY(&lights[curr_light].dir, &presult);
    } // end for

    } break;

 case TRANSFORM_TRANS_ONLY:
    {
    // 遍历所有的光源
    for (curr_light = 0; curr_light < num_lights; curr_light++)
      {
      // 对变换后的坐标进行变换
      POINT4D presult; // 用于存储每次变换的结果

      // 对光源的位置进行变换
      Mat_Mul_VECTOR4D_4X4(&lights[curr_light].tpos, mt, &presult);

      // 将结果存回去
      VECTOR4D_COPY(&lights[curr_light].tpos, &presult);

      // 对方向向量进行变换
      Mat_Mul_VECTOR4D_4X4(&lights[curr_light].tdir, &mr, &presult);

      // 将结果存回去
      VECTOR4D_COPY(&lights[curr_light].tdir, &presult);
      } // end for

     } break;

 case TRANSFORM_LOCAL_TO_TRANS:
    {
    // 遍历所有的光源
    for (curr_light = 0; curr_light < num_lights; curr_light++)
    {
    // 对局部/世界坐标进行变换，将结果存储到 tpos 和 tdir 中
    // 这是最常见的调用该函数的方式
    POINT4D presult; // 用于存储变换结果

    // 对光源的位置进行变换
    Mat_Mul_VECTOR4D_4X4(&lights[curr_light].pos, mt,
              &lights[curr_light].tpos);

    // 对方向向量进行变换
    Mat_Mul_VECTOR4D_4X4(&lights[curr_light].dir, &mr,
```

```
                &lights[curr_light].tdir);
      } // end for

      } break;

    default: break;

    } // end switch

} // end Transform_LIGHTSV2
```

调用该函数时，需要指定光源列表、光源数、变换矩阵以及对哪种坐标进行变换。该函数执行标准变换，与读者之前看到的变换函数的工作原理相同。在 99%的情况下，将像下面这样调用它：

```
Transform_LIGHTSV2(lights2, 4, &cam.mcam, TRANSFORM_LOCAL_TO_TRANS);
```

上述调用使用矩阵 cam.mcam 对数组 light2[ ]中的光源进行变换，有 4 个光源需要变换；对存储在 pos 和 dir 中的坐标进行变换，并将结果存储在 tpos 和 tdir 中。创建新的光源数据结构后，在 T3DLIB8.CPP|H 中声明了一个存储光源的新数组：

```
LIGHTV2 lights2[MAX_LIGHTS]; //系统中的光源
```

另外，在头文件中，定义了一些针对新光源数据结构的常量：

```
// 为 2.0 版光源数据结构定义的常量
#define LIGHTV2_ATTR_AMBIENT     0x0001    // 环境光源
#define LIGHTV2_ATTR_INFINITE    0x0002    // 无穷远光源
#define LIGHTV2_ATTR_DIRECTIONAL  0x0002    // 无穷远光源（别名）
#define LIGHTV2_ATTR_POINT       0x0004    // 点光源
#define LIGHTV2_ATTR_SPOTLIGHT1    0x0008    // 1 类聚光灯
#define LIGHTV2_ATTR_SPOTLIGHT2    0x0010    // 2 类聚光灯

#define LIGHTV2_STATE_ON        1        // 开启
#define LIGHTV2_STATE_OFF       0        // 关闭

// 变换控制标志
#desine TRANSFORM_COPY_LOCAL_TO_TRANS 3  // 将 POS 和 dir 中的值复制到 tpos 和 tdir 中，不进行任何变换
```

这些常量的值与 LIGHTV1 版相同，只是名称前缀从 LIGHTV1 变成了 LIGHTV2。另外，还有一个专门为函数 Transform_LIGHTSV2()定义的常量：TRANSFORM_COPY_TO_TRANS，它指定不进行任何变换，而只是将 pos 和 dir 中的值复制到 tpos 和 tdir 中。

最后，必须重新编写所有的光照函数。然而除了名称不同以及执行光照计算时使用 tpos 和 tdir 而不是 pos 和 dir 中的值外，其他方面几乎没有变化。这些函数太大，这里不列出它们的代码，而只列出它们的原型（位于头文件中）：

```
int Light_OBJECT4DV2_World2_16(OBJECT4DV2_PTR obj,  // 要处理的物体
     CAM4DV1_PTR cam,    // 相机
     LIGHTV2_PTR lights, // 光源列表（可能有多个）
     int max_lights);   // 列表中最大光源数

int Light_OBJECT4DV2_World2(OBJECT4DV2_PTR obj,  // 要处理的物体
     CAM4DV1_PTR cam,    // 相机
     LIGHTV2_PTR lights, // 光源列表（可能有多个）
     int max_lights);   // 列表中最大光源数

int Light_RENDERLIST4DV2_World2( RENDERLIST4DV2_PTR rend_list,
                      // 要处理的渲染列表
     CAM4DV1_PTR cam,     // 相机
     LIGHTV2_PTR lights,  // 光源列表（可能有多个）
     int max_lights);     // 列表中最大光源数

int Light_RENDERLIST4DV2_World2_16(RENDERLIST4DV2_PTR rend_list,  //要处理的渲染列表
     CAM4DV1_PTR cam,    // 相机
     LIGHTV2_PTR lights, // 光源列表（可能有多个）
     int max_lights);   // 列表中最大光源数
```

```
// 光照系统
int Init_Light_LIGHTV2(LIGHTV2_PTR lights,      // 使用的光源数组（新增的）
     int      index,// 要创建的光源的索引（0 到 MAX_LIGHTS-1）
     int     _state,// 光源状态
     int     _attr, // 光源类型和其他属性
     RGBAV1    _c_ambient, // 环境光强度
     RGBAV1    _c_diffuse, // 散射光强度
     RGBAV1    _c_specular,// 镜面反射光强度
     POINT4D_PTR _pos,    // 光源的位置
     VECTOR4D_PTR _dir,   // 光源的方向
     float     _kc,           // 衰减因子
     float     _kl,
     float     _kq,
     float     _spot_inner, // 聚光灯的内锥角
     float     _spot_outer, // 聚光灯的外锥角
     float     _pf);     // 聚光灯的指数因子

int Reset_Lights_LIGHTV2(LIGHTV2_PTR lights,  // 使用的光源数组（新增的）
             int max_lights);   // 系统中的光源数
```

与前一个版本相比，这些函数惟一不同的地方是，第一个参数是新增的，它是一个指向光源的指针。这让光照系统可以有多组全局光源。

警告：指定光源的位置和方向时，务必使用最后一个分量为 1.0 的点（POINT4D）和向量（VECTOR4D）。否则，当光源被变换时，变换矩阵中的平移部分将无效。

有关调用这些新函数的范例，请参阅演示程序。重新编写光照函数旨在能够在相机坐标空间中执行光照计算，这样便可以将光照计算推迟到多边形数量最少时进行。

10.4　地 形 小 议

进行 3D 裁剪的原因很多，但最主要的原因是最大限度地减少传递给流水线下游的多边形数量，同时避免对 z 坐标小于或等于 0 的多边形进行投影，以免给透视计算带来问题。到目前为止，本书创建的演示程序都是室外的、基于物体的，没有大型多边形和室内环境，因此在很大程度上说不会出现上述情况。然而，我们还不能加载各种游戏关卡、室内环境模型和大型多边形游戏世界，因此无法演示这种问题。作者决定创建一个粗糙的地形引擎函数，它能够生成非常大的地形，然后根据一个高程图（height map）将其分割为三角形。

这存在两个问题。首先，除 OBJECT4DV2 外，我们还没有其他表示 3D 实体的结构，而 OBJECT4DV2 并非设计用于存储大型网格的，但仍可以暂时使用它。然而，真正的问题在于，地形由数千个甚至数百万个多边形组成。不过，通过使用分区算法（sectorizing algorithm）可以剔除 99%的多边形；这种算法通过拼贴地形片（terrain patch）来创建大型地形，然后只处理可见的地形片。实际上，每个地形片都是一个物体，这样将有一个大型地形片数组，如图 10.24 所示。

可以将多个物体拼贴起来，以创建大型地形，然后使用物体剔除操作来剔除不可见的地形片。但另一个更严峻的问题是，即使是单个地形片，也可能包含 32×32、64×64 或 256×256 个高度值，这意味着它包含 31×31、63×63 或 255×255 个分片（tile），因为在每一维中，分片数总是比高度值数目少 1。这是因为我们根据高度值来确定包围分片的四边形的顶点，因此每个地形片包含 31^2*2、63^2*2 或 255^2*2 个三角形。

即使进行裁剪和剔除，这么多的三角形也是承受不了的。实际的地形引擎动态地执行详细程度（level-of-detail）分析，将不需要的多边形进行合并，在需要时再将它们分割。观察单个地形片时，如果它包含 64×64 个分片，每个分片由两个三角形组成，模型中将有 8192 个三角形；但经过简化后，实际上可能只需要几百个。这里不打算介绍这种算法，因为它属于地形渲染和建模的范畴。

接下来介绍如何创建简单的地形。只需调用一个函数就可以生成这种地形，该函数将地形存储到物体中，我们可以像处理物体那样处理它。

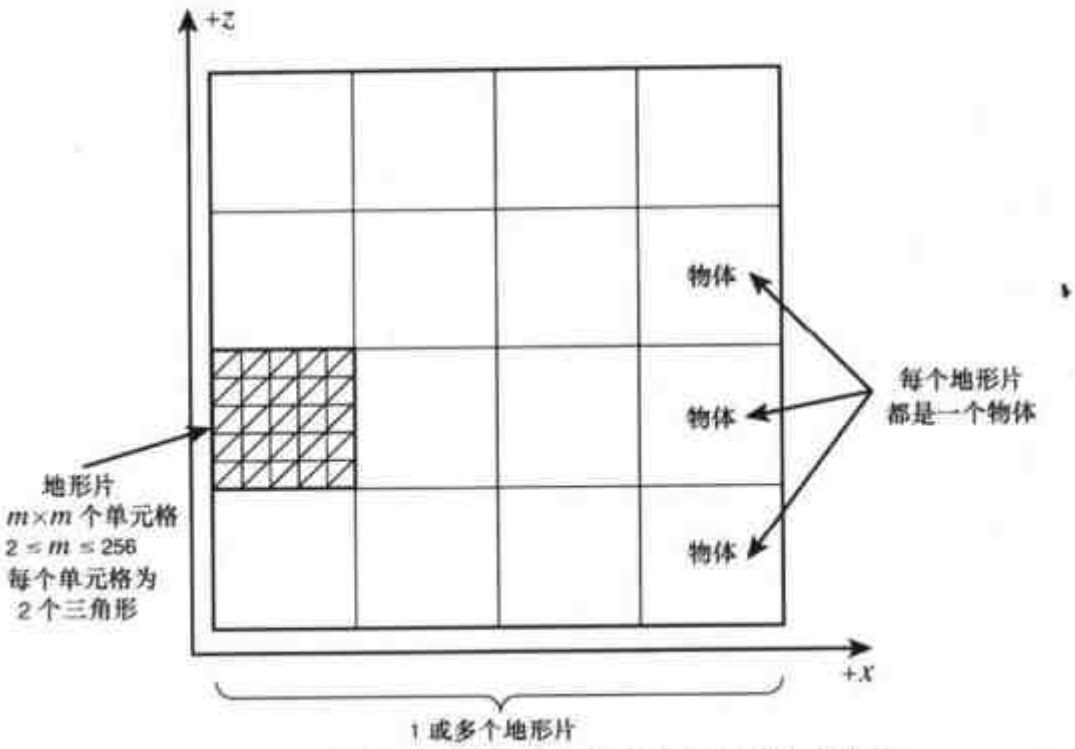


图 10.24　通过将地形划分为物体来创建大型游戏世界

10.4.1　地形生成函数

生成地形的方法有很多：使用随机数；基于卫星数据，使用颜色高程图，组合使用这些方法等。我们将使用高程图来生成地形。我们的进攻计划是，加载一个 256 色的位图，它表示要模拟的高程场（height field）；同时将一个彩色纹理映射到高程场上，以显示地形。图 10.25 是一个这样的高程场，为让读者能够看清楚，该高程场被放大了很多倍。高程图中的颜色表示高度值。

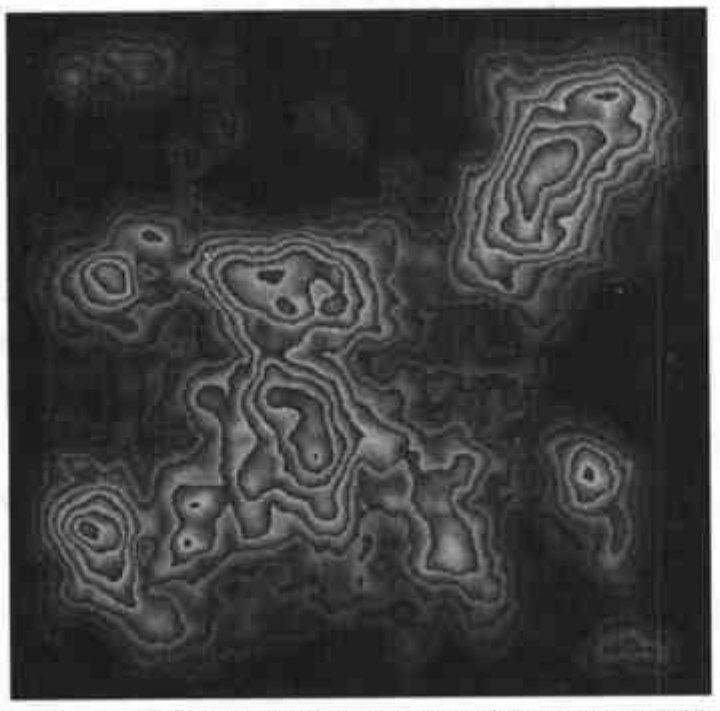


图 10.25　将位图中的颜色索引来用作地形的高度数据

以这样的方式解释高程图：生成一个以世界坐标系原点(0，0，0)为中心的规则栅格（grid），其中每个单元格顶点处的高度与位图中相应的像素值成正比。例如，颜色索引 0 对应的高度最小，255 对应的高度最大。可以将颜色索引值乘以一个缩放因子，来得到所需的高度范围。例如，图 10.26 是根据一个 4×4 的位图生成的高程场，其中标出了每个顶点处的高度值。

根据数据生成高程场网格只需要使用两个 for 循环，同时要注意“少 1”误差。该算法将每个单元格划分成两个三角形，单元格顶点处的高度是根据颜色索引和缩放因子计算得到的。另外，还需要知道高程场的大小，即其宽度和高度（读者可能想称之为宽度和长度）是多少。我们将在 x-z 平面上生成高程场，并根据位图数据来确定高程场中每个单元格顶点处的高度。然后将每个单元格分割成两个三角形，并将它们插入到物体的顶点列表中。

前面提到过“少 1”误差。这是什么意思呢？0～10 有多少个整数？11 个。绘制 5 条直线时，将划分出多少个停车位？4 个。这就是“少 1”误差——需要确保计数是正确的。人工生成栅格时这很重要，如果没有减 1，可能浪费内存或导致异常。作者喜欢使用较小的数组来验证算法。通常先验证 1×1 的情况，然后验证 4×4 的情况。验证在 1 和 n 的情况下算法成立后，便可根据归纳法确定所有情况下都是成立的。

手工创建物体也不算太糟糕，只需提供要生成的物体的信息即可，而这些信息原本是从磁盘中读取的。需要顶点列表、多边形列表、顶点数和多边形数以及每个多边形的属性。除多边形属性（光照信息和纹理信息）外，其他数据都很简单。必须手工将光照信息和纹理信息传递给地形生成函数，为此定义了下述常量：

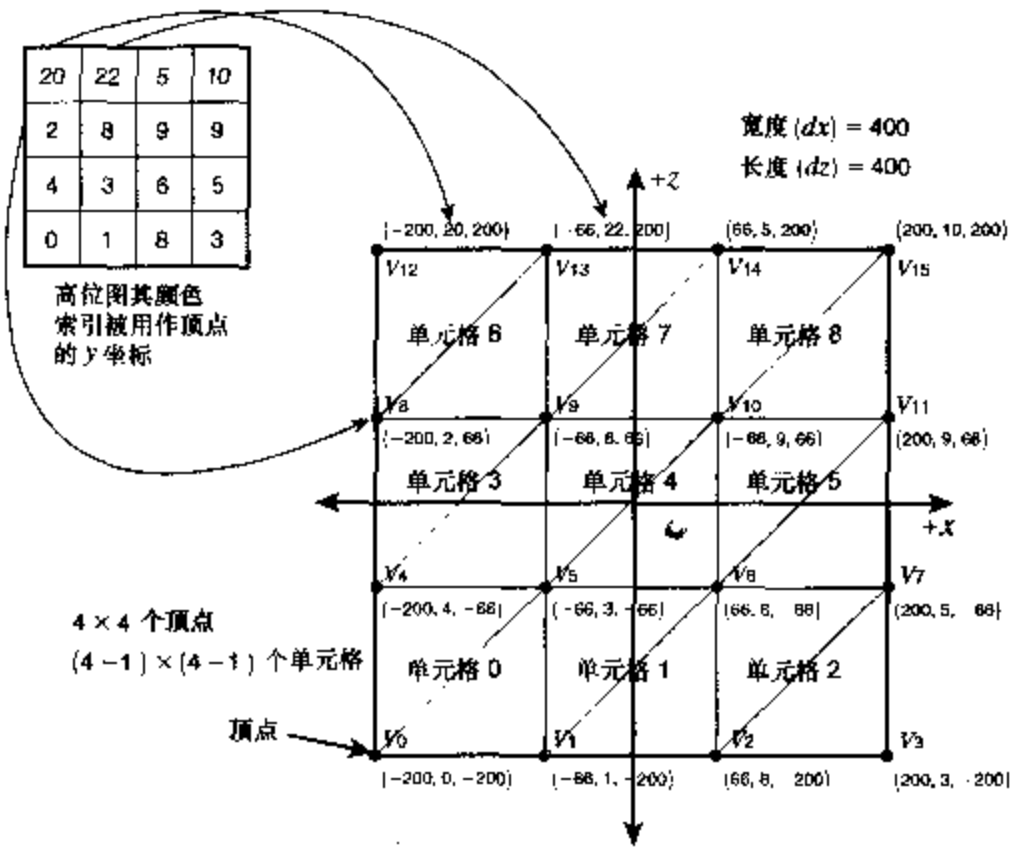


图 10.26　一个 4×4 的高程场

```
// 为 2.0 版多边形和多边形面定义的常量

// 多边形和多边形面的属性
#define POLY4DV2_ATTR_2SIDED          0x0001
#define POLY4DV2_ATTR_TRANSPARENT     0x0002
#define POLY4DV2_ATTR_8BITCOLOR       0x0004
#define POLY4DV2_ATTR_RGB16           0x0008
#define POLY4DV2_ATTR_RGB24           0x0010
#define POLY4DV2_ATTR_SHADE_MODE_PURE    0x0020
#define POLY4DV2_ATTR_SHADE_MODE_CONSTANT  0x0020 // 别名
#define POLY4DV2_ATTR_SHADE_MODE_EMISSIVE  0x0020 // 别名

#define POLY4DV2_ATTR_SHADE_MODE_FLAT    0x0040
#define POLY4DV2_ATTR_SHADE_MODE_GOURAUD  0x0080
#define POLY4DV2_ATTR_SHADE_MODE_PHONG   0x0100
#define POLY4DV2_ATTR_SHADE_MODE_FASTPHONG 0x0100 // 别名
#define POLY4DV2_ATTR_SHADE_MODE_TEXTURE  0x0200

// 新增的
#define POLY4DV2_ATTR_ENABLE_MATERIAL  0x0800 // 根据材质进行光照计算
#define POLY4DV2_ATTR_DISABLE_MATERIAL 0x1000 // 根据原来的颜色进行光照计算
                    // 模拟 1.0 版本
```

其中以粗体显示的属性常量是我们关心的。下面花些时间来讨论它们。地形生成函数可在 8 位或 16 位模式下生成地形，调用它时必须使用下述常量来指定模式：

```
POLY4DV2_ATTR_8BITCOLOR
POLY4DV2_ATTR_RGB16
```

该函数支持固定、恒定和 Gouraud 着色，从几何上说，它们之间没有任何差别，只是使用 Gouraud 着色时需要计算法线。但无需考虑这一点，因为创建网格后，我们将调用下述函数来计算法线：

```
// 计算多边形法线的长度
Compute_OBJECT4DV2_Poly_Normals(obj);

// 为使用 Gouraud 着色的多边形计算顶点法线
Compute_OBJECT4DV2_Vertex_Normals(obj);
```

它们分别计算多边形法线（供背面消除和多边形光照计算使用）和顶点法线（供 Gouraud 着色使用）。要设置着色模式，可使用下述常量之一：

```
POLY4DV2_ATTR_SHADE_MODE_EMISSIVE
```

```
POLY4DV2_ATTR_SHADE_MODE_FLAT
POLY4DV2_ATTR_SHADE_MODE_GOURAUD
```

如果要将纹理映射到地形表面，该如何办呢？需要使用下面的常量启用纹理映射：

```
POLY4DV2_ATTR_SHADE_MODE_TEXTURE
```

然而，在纹理映射被启用时，光栅化函数只支持固定和恒定着色，因此不能采用 Gouraud 着色处理。例如，要在 16 位模式下创建一个使用 Gouraud 着色的地形，可使用下述常量：

```
POLY4DV2_ATTR_RGB16 | POLY4DV2_ATTR_SHADE_MODE_GOURAUD
```

要创建一个使用恒定处理且带纹理的地形，可使用下述常量：

```
POLY4DV2_ATTR_RGB16 |
POLY4DV2_ATTR_SHADE_MODE_FLAT
|POLY4DV2_ATTR_SHADE_MODE_TEXTURE
```

正如读者看到的，只需使用逻辑 OR 运算将属性标记组合起来即可。接下来讨论对纹理映射的支持。

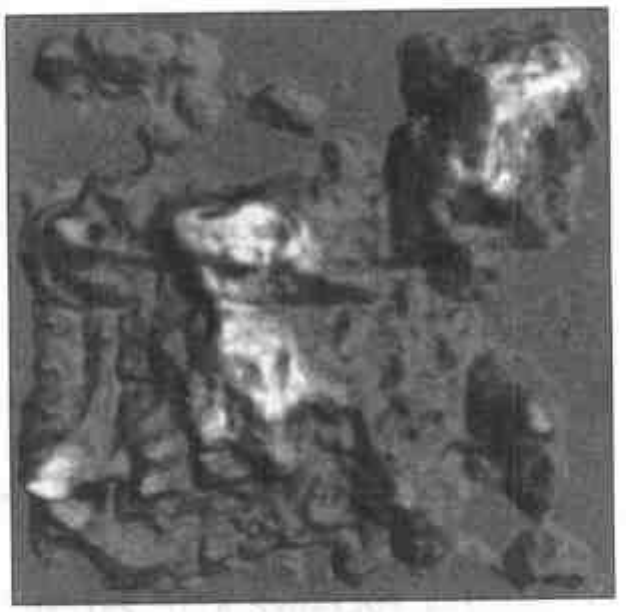


图 10.27　一个用于地形的纹理图

图 10.27 是一个位图，可将其作为纹理映射到地形表面上；它实际上是由程序生成的一个用于地形的纹理图，因此将其映射到地图表面时效果会很不错。这个位图是 256×256 的，但也可以是任何 2 的幂，最小可以为 88 像素的。在我们的系统中，光栅化函数支持的最大纹理图为 256×256。

作者采用的方法是，计算输入纹理图的纹理坐标，然后将纹理映射到地形图上。当然，可以使用一个大型纹理图，将其分割成 256×256 的小块，然后将它们映射到多边形上。但就现在而言，前一种方法更简单，因为作者并非要编写一个真正的地形引擎，这里编写地形生成函数只是为了创建裁剪演示程序。总之，要使用纹理，必须将纹理图的文件名（以及提供高度信息的 256 色位图的文件名）传递给地形生成函数，该函数将计算纹理坐标，以便将纹理映射到地图表面。这很简单，只是涉及一些细节：地形图的宽度、长度和单元格数。

因此，要对地形进行纹理映射，必须传递一个 16 位纹理图的文件名。当然，纹理图应是方形的。地形生成函数将计算地形中每个三角形的纹理坐标，以便将纹理映射到地形表面。

在大多数情况下，地形生成函数接受两个输入文件：一个是.BMP 格式的 256 色高程图，它应尽可能为方形，通常小于 40×40 像素。这是因为更大的高程图将导致地形的高度值数量超过 40×40 个（四边形数量超过 39×39 个，三角形数量超过 $39^2*2 = 3042$ 个）。对于小型软件引擎而言，这是极限了。

接下来介绍函数代码，并通过几个范例演示如何调用它。

```
int Generate_Terrain_OBJECT4DV2(OBJECT4DV2_PTR obj,     // 指向物体的指针
 float twidth,       // 世界坐标系中的宽度（x 轴）
 float theight,      // 世界坐标系中的长度（z 轴）
 float vscale,       // 最大可能高度值
 char *height_map_file,  // 256 色高程图的文件名
 char *texture_map_file, // 纹理图的文件名
 int rgbcolor,       // 没有纹理时地形的颜色
 VECTOR4D_PTR pos,   // 初始位置
 VECTOR4D_PTR rot,   // 初始旋转角度
 int poly_attr)    // 着色属性
{
// 这个函数生成一个在 x-z 平面上为 width × height 的地形
// 地形的高度值由 256 色的高程图定义，最大可能高度值为 vscale(对应于颜色索引 255)
// 高程图中的每个像素对应于地形中一个单元格顶点，而每个单元格被划分成两个三角形
// 因此，如果高程图大小为 256×256，将有(256-1) x (256-1) 个单元格
// 另外，如果指定了纹理图，它将被映射到地形上，因此将生成纹理坐标
// 这个函数生成 8 位或 16 位的地形，因此调用它时必须指定正确的属性，使其与纹理图位深匹配

char buffer[256];  // 工作缓冲区

float col_tstep, row_tstep;
float col_vstep, row_vstep;
```

```
int columns, rows;

int rgbwhite;

BITMAP_FILE height_bitmap; // 存储高程图

// 第 1 步：清空物体的数据并进行初始化
memset(obj, 0, sizeof(OBJECT4DV2));

// 将物体的状态设置为活动的和可见的
obj->state = OBJECT4DV2_STATE_ACTIVE | OBJECT4DV2_STATE_VISIBLE;

// 设置物体的位置
obj->world_pos.x = pos->x;
obj->world_pos.y = pos->y;
obj->world_pos.z = pos->z;
obj->world_pos.w = pos->w;

// 根据指定的地形位深创建合适的颜色字（color word）
// rgbcolor 总是为 RGB 5.6.5 格式，因此仅当位深为 8 位时才需要向下转换
if (poly_attr & POLY4DV1_ATTR_8BITCOLOR)
  {
  rgbcolor = rgblookup[rgbcolor];
  rgbwhite = rgblookup[RGB16Bit(255,255,255)];
  } // end if
else
  {
  rgbwhite = RGB16Bit(255,255,255);
  } // end else

// 设置帧数
obj->num_frames = 1;
obj->curr_frame = 0;
obj->attr = OBJECT4DV2_ATTR_SINGLE_FRAME;

// 清空位图
memset(&height_bitmap, 0, sizeof(BITMAP_FILE));
memset(&bitmap16bit, 0, sizeof(BITMAP_FILE));

// 第 2 步：加载高程图
Load_Bitmap_File(&height_bitmap, height_map_file);

// 计算基本信息
columns = height_bitmap.bitmapinfoheader.biWidth;
rows    = height_bitmap.bitmapinfoheader.biHeight;

col_vstep = twidth / (float)(columns - 1);
row_vstep = theight / (float)(rows - 1);

sprintf(obj->name ,"Terrain:%s%s", height_map_file, texture_map_file);
obj->num_vertices = columns * rows;
obj->num_polys    = ((columns - 1) * (rows - 1) ) * 2;

// 存储一些结果，供地形跟踪算法使用
obj->ivar1 = columns;
obj->ivar2 = rows;
obj->fvar1 = col_vstep;
obj->fvar2 = row_vstep;

// 分配存储顶点列表和多边形列表的空间
if (!Init_OBJECT4DV2(obj,  // object to allocate
          obj->num_vertices,
          obj->num_polys,
          obj->num_frames))
 {
 Write_Error("\nTerrain generator error (can't allocate memory).");
 } // end if
```

```
// 加载纹理图(如果有的话)
if ( (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE) && texture_map_file)
  {
  // 从磁盘中加载纹理图
  Load_Bitmap_File(&bitmap16bit, texture_map_file);

  // 创建一个大小和位深合适的位图
  obj->texture = (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE));
  Create_Bitmap(obj->texture,0,0,
        bitmap16bit.bitmapinfoheader.biWidth,
        bitmap16bit.bitmapinfoheader.biHeight,
        bitmap16bit.bitmapinfoheader.biBitCount);

  // 加载位图图像
  if (obj->texture->bpp == 16)
    Load_Image_Bitmap16(obj->texture,
       &bitmap16bit,0,0,BITMAP_EXTRACT_MODE_ABS);
  else
    {
    Load_Image_Bitmap(obj->texture,
       &bitmap16bit,0,0,BITMAP_EXTRACT_MODE_ABS);
    } // end else 8 bit

  // 计算纹理图的步进因子（stepping factor），供计算纹理坐标时使用
  col_tstep = (float)(bitmap16bit.bitmapinfoheader.biWidth-1) /
        (float)(columns - 1);
  row_tstep = (float)(bitmap16bit.bitmapinfoheader.biHeight-1) /
       (float)(rows - 1);

  // 指出物体带纹理
  SET_BIT(obj->attr, OBJECT4DV2_ATTR_TEXTURES);

  // 卸载位图
  Unload_Bitmap_File(&bitmap16bit);
  } // end if

Write_Error("\ncolumns = %d, rows = %d", columns, rows);
Write_Error("\ncol_vstep = %f, row_vstep = %f", col_vstep, row_vstep);
Write_Error("\ncol_tstep=%f, row_tstep=%f", col_tstep, row_tstep);
Write_Error("\nnum_vertices = %d, num_polys = %d", obj->num_vertices,
      obj->num_polys);

//第 4 步：按先行后列的顺序生成顶点列表和纹理坐标列表
for (int curr_row = 0; curr_row < rows; curr_row++)
  {
  for (int curr_col = 0; curr_col < columns; curr_col++)
    {
    int vertex = (curr_row * columns) + curr_col;
    // 计算顶点坐标
    obj->vlist_local[vertex].x = curr_col * col_vstep - (twidth/2);
    obj->vlist_local[vertex].y = vscale*((float)
    height_bitmap.buffer[curr_col + (curr_row * columns) ]) / 255;
    obj->vlist_local[vertex].z = curr_row * row_vstep - (theight/2);

    obj->vlist_local[vertex].w = 1;

    // 每个顶点都至少包含一个点，相应地设置其属性
    SET_BIT(obj->vlist_local[vertex].attr, VERTEX4DTV1_ATTR_POINT);

    // 需要纹理坐标?
    if ( (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE) && texture_map_file)
      {
      // 计算纹理坐标
      obj->tlist[vertex].x = curr_col * col_tstep;
      obj->tlist[vertex].y = curr_row * row_tstep;
```

```
    } // end if

   Write_Error("\nVertex %d: V[%f, %f, %f], T[%f, %f]",
          vertex, obj->vlist_local[vertex].x,
          obj->vlist_local[vertex].y,
          obj->vlist_local[vertex].z,
          obj->tlist[vertex].x,
          obj->tlist[vertex].y);
   } // end for curr_col

  } // end curr_row

// 计算平均半径和最大半径
Compute_OBJECT4DV2_Radius(obj);

Write_Error("\nObject average radius = %f, max radius = %f",
     obj->avg_radius[0], obj->max_radius[0]);

// 第 5 步：生成多边形列表
for (int poly=0; poly < obj->num_polys/2; poly++)
  {
  // 每个单元格有两个三角形，单元格顶点按先行后列的顺序排列
  // 如果顶点数组大小为 mxn，则多边形列表大小为 2x(m-1)x(n-1)

  int base_poly_index = (poly % (columns-1)) +
            (columns * (poly / (columns - 1)) );

  // 当前单元格的左下多边形
  obj->plist[poly*2].vert[0] = base_poly_index;
  obj->plist[poly*2].vert[1] = base_poly_index+columns;
  obj->plist[poly*2].vert[2] = base_poly_index+columns+1;

  //当前单元格的右上多边形
  obj->plist[poly*2+1].vert[0] = base_poly_index;
  obj->plist[poly*2+1].vert[1] = base_poly_index+columns+1;
  obj->plist[poly*2+1].vert[2] = base_poly_index+1;

  // 将多边形顶点列表指向物体的顶点列表
  obj->plist[poly*2].vlist = obj->vlist_local;
  obj->plist[poly*2+1].vlist = obj->vlist_local;

  // 根据参数 poly_attr 相应地设置多边形的属性
  obj->plist[poly*2].attr = poly_attr;
  obj->plist[poly*2+1].attr = poly_attr;

  // 设置多边形的颜色
  obj->plist[poly*2].color = rgbcolor;
  obj->plist[poly*2+1].color = rgbcolor;

  // 检查着色方法是否是 Gouraud 或 Phong，如果是，则需要顶点法线
  if ( (obj->plist[poly*2].attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD) ||
     (obj->plist[poly*2].attr & POLY4DV2_ATTR_SHADE_MODE_PHONG) )
    {
    // 设置顶点的属性，指出它包含法线
    SET_BIT(obj->vlist_local[ obj->plist[poly*2].vert[0] ].attr,
        VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[ obj->plist[poly*2].vert[1] ].attr,
        VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[ obj->plist[poly*2].vert[2] ].attr,
        VERTEX4DTV1_ATTR_NORMAL);

    SET_BIT(obj->vlist_local[ obj->plist[poly*2+1].vert[0] ].attr,
        VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[ obj->plist[poly*2+1].vert[1] ].attr,
        VERTEX4DTV1_ATTR_NORMAL);
    SET_BIT(obj->vlist_local[ obj->plist[poly*2+1].vert[2] ].attr,
```

```
        VERTEX4DTV1_ATTR_NORMAL);

    } // end if

    // 如果启用了纹理映射，则计算纹理坐标
    if (poly_attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
     {
     // 指定多边形使用的纹理
     obj->plist[poly*2].texture = obj->texture;
     obj->plist[poly*2+1].texture = obj->texture;

     // 设置纹理坐标
     // 左下三角形
     obj->plist[poly*2].text[0] = base_poly_index;
     obj->plist[poly*2].text[1] = base_poly_index+columns;
     obj->plist[poly*2].text[2] = base_poly_index+columns+1;

     // 右上三角形
     obj->plist[poly*2+1].text[0] = base_poly_index;
     obj->plist[poly*2+1].text[1] = base_poly_index+columns+1;
     obj->plist[poly*2+1].text[2] = base_poly_index+1;

     // 重新设置多边形颜色，使其反射率更高
     obj->plist[poly*2].color = rgbwhite;
     obj->plist[poly*2+1].color = rgbwhite;

     // 设置纹理坐标属性
     SET_BIT(obj->vlist_local[ obj->plist[poly*2].vert[0] ].attr,
                  VERTEX4DTV1_ATTR_TEXTURE);
     SET_BIT(obj->vlist_local[ obj->plist[poly*2].vert[1] ].attr,
                  VERTEX4DTV1_ATTR_TEXTURE);
     SET_BIT(obj->vlist_local[ obj->plist[poly*2].vert[2] ].attr,
                  VERTEX4DTV1_ATTR_TEXTURE);
       SET_BIT(obj->vlist_local[ obj->plist[poly*2+1].vert[0] ].attr,
                  VERTEX4DTV1_ATTR_TEXTURE);
       SET_BIT(obj->vlist_local[ obj->plist[poly*2+1].vert[1] ].attr,
                  VERTEX4DTV1_ATTR_TEXTURE);
       SET_BIT(obj->vlist_local[ obj->plist[poly*2+1].vert[2] ].attr,
                  VERTEX4DTV1_ATTR_TEXTURE);

       } // end if

     // 将材质模式设置为不使用材质
     SET_BIT(obj->plist[poly*2].attr, POLY4DV2_ATTR_DISABLE_MATERIAL);
     SET_BIT(obj->plist[poly*2+1].attr, POLY4DV2_ATTR_DISABLE_MATERIAL);

     // 将三角形的状态设置为活动的
     obj->plist[poly*2].state = POLY4DV2_STATE_ACTIVE;
     obj->plist[poly*2+1].state = POLY4DV2_STATE_ACTIVE;

     // 将多边形顶点列表指向物体的顶点列表
     obj->plist[poly*2].vlist = obj->vlist_local;
     obj->plist[poly*2+1].vlist = obj->vlist_local;

     // 设置纹理坐标列表
     obj->plist[poly*2].tlist = obj->tlist;
     obj->plist[poly*2+1].tlist = obj->tlist;

    } // end for poly
#if 0
for (poly=0; poly < obj->num_polys; poly++)
{
Write_Error("\nPoly %d: Vi[%d, %d, %d], Ti[%d, %d, %d]",poly,
      obj->plist[poly].vert[0],
      obj->plist[poly].vert[1],
```

```
        obj->plist[poly].vert[2],
        obj->plist[poly].text[0],
        obj->plist[poly].text[1],
        obj->plist[poly].text[2]);

} // end
#endif

// 计算多边形法线长度
Compute_OBJECT4DV2_Poly_Normals(obj);

// 为使用 Gouraud 着色的多边形计算顶点法线
Compute_OBJECT4DV2_Vertex_Normals(obj);

// 成功返回
return(1);

} // end Generate_Terrain_OBJECT4DV2
```

这个函数是在.PLG 加载函数的基础上修改而成的。由于.PLG 加载函数提供了创建 OBJECT4DV2 的模板，因此以它为基础，加入人工生成地形的算法。接下来看看如何调用该函数。下面的例子生成一个 32×32 的地形，地形的大小为 4000×4000，它带纹理并使用恒定着色，使用的高程图文件为 EARTHEHEIGHTMAP01.BMP，纹理图是 256×256×16 的，文件名为 EARTHCOLORMAP01.BMP：

```
VECTOR4D terrain_pos = {0,0,0,0};

Generate_Terrain_OBJECT4DV2(&obj_terrain, // 指向物体的指针
 4000,    // 地形在世界坐标空间中的宽度(x 轴)
 4000,    // 地形在世界坐标空间中的长度(z 轴)
 700,     // 可能的最大高度值
 "earthheightmap01.bmp", // 256 色高程图的文件名
 "earthcolormap01.bmp", // 纹理图的文件名
 RGB16Bit(255,255,255), // 没有纹理时地形的颜色
 &terrain_pos,       // 初始位置
 NULL,          // 初始旋转角度
 POLY4DV2_ATTR_RGB16 |
 POLY4DV2_ATTR_SHADE_MODE_FLAT | POLY4DV2_ATTR_SHADE_MODE_TEXTURE);
```

这个函数的参数很简单。您指定地形在世界坐标系中的宽度和长度（*x*-*z* 边界，这里为 4000×4000）以及最大可能高度值（这里为 770）。接下来是磁盘中高程图和纹理图的名称（如果不使用纹理，应将纹理图的文件名设置为 NULL）；然后是地形的颜色（不使用纹理时，必须告诉函数生成什么颜色的地形）。接下来是地形在世界坐标系中的位置（通常(0，0，0)是不错的选择）；然后是地形的初始旋转角度；最后是多边形属性（这里为恒定着色、16 位模式、使用纹理映射）。

可以创建任意数量的地形物体，如果愿意，可以创建一个大型游戏世界，例如包含 16×16 个地形，每个都是一个物体，然后利用物体剔除功能。但正如前面指出的，作者只是想提供可用于创建大型多边形的东西，以模拟室内或室外环境，而地形物体很管用。

10.4.2　生成地形数据

读者可能会问，如何生成地形数据呢？为此，可创建一个程序。另外，还必须确保用于地形的纹理是合理的。例如，积雪应该在山顶，而水应该是蓝色的。然而，大多数人使用这样的程序来生成地形：它能够生成.BMP 格式的 256 色高程图以及将被映射到地形表面的纹理图。一个这样的程序是 VistaPro，其最新版本为 4.0。问题是，任何人都有权修改这个程序，因此最好的办法是，在网上搜索“VistaPro”，找到最新的发行人。另一个可生成地形和高程图的程序是 MetaCreations 的 Bryce，该公司已被 Corel 兼并，因此 Bryce 归 Corel 所有。网上的地形生成程序数不胜数，这里只列出了其中的两个。

10.4.3　沙地汽车演示程序

现在来使用地形纹理映射和裁剪功能，看它们是否管用。为此，作者创建了一个名为

DEMOII10_2.CPP|EXE（8 位版本为 DEMOII10_2_8b.CPP|EXE）的小型演示程序，来模拟荒岛上的沙地汽车。图 10.28 和 10.29 是该演示程序的屏幕截图。要编译该演示程序，需要使用主.CPP 文件、文件 T3DLIB1.CPP|H 到 T3DLIB8.CPP|H 以及标准 DirectX .LIB 文件。

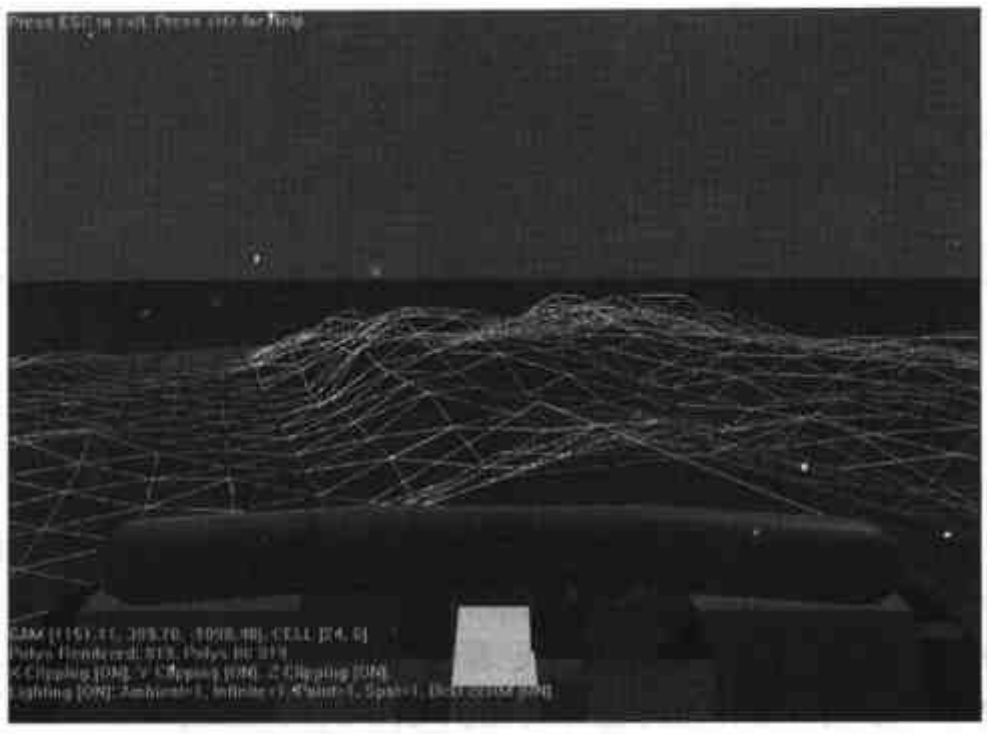


图 10.28　线框模式下地形演示程序的屏幕截图



图 10.29　实体模式下使用纹理映射时地形演示程序的屏幕截图

和以往一样，这个演示程序也非常简单。它调用一个函数来生成地形，使用的参数值与前面的函数调用范例中相同；然后通过改变相机位置，让玩家在地形上驾驶沙地汽车。后面将简要地介绍如何跟踪地形。

按 H 键可以显示/关闭帮助信息，其中指出了如何控制演示程序。最重要的是，玩家可以按 X、Y、Z 键来启用/禁用根据 *x*、*y*、*z* 裁剪面进行的 3D 裁剪，然后驾驶沙地汽车，并查看屏幕下面的多边形状态。启用/禁用 *x*、*y*、*z* 裁剪时，请注意流水线状态的变化。

警告：如果禁用根据近裁剪面进行 3D 裁剪，可能导致应用程序崩溃，这是因为在这种情况下，穿过近裁剪面的大型多边形未经裁剪就被投影。这很糟糕，但当然不会损坏计算机。最后，可使用 W 键切换到线框模式，以便能够看到所有的多边形。

地形跟踪算法

地形跟踪（following）算法和虚拟沙地汽车的物理模型是彼此相关的，这里一并介绍它们。问题是这样的：我们使用相机来表示玩家的视点（当然，相机前面是预先渲染好的小型沙地汽车）。玩家驾驶的沙地汽车必须在地形表面，而不能穿入地面，因此需要计算相机所在位置处的地形高度，并将视点相应地往上

移，如图 10.30 所示。

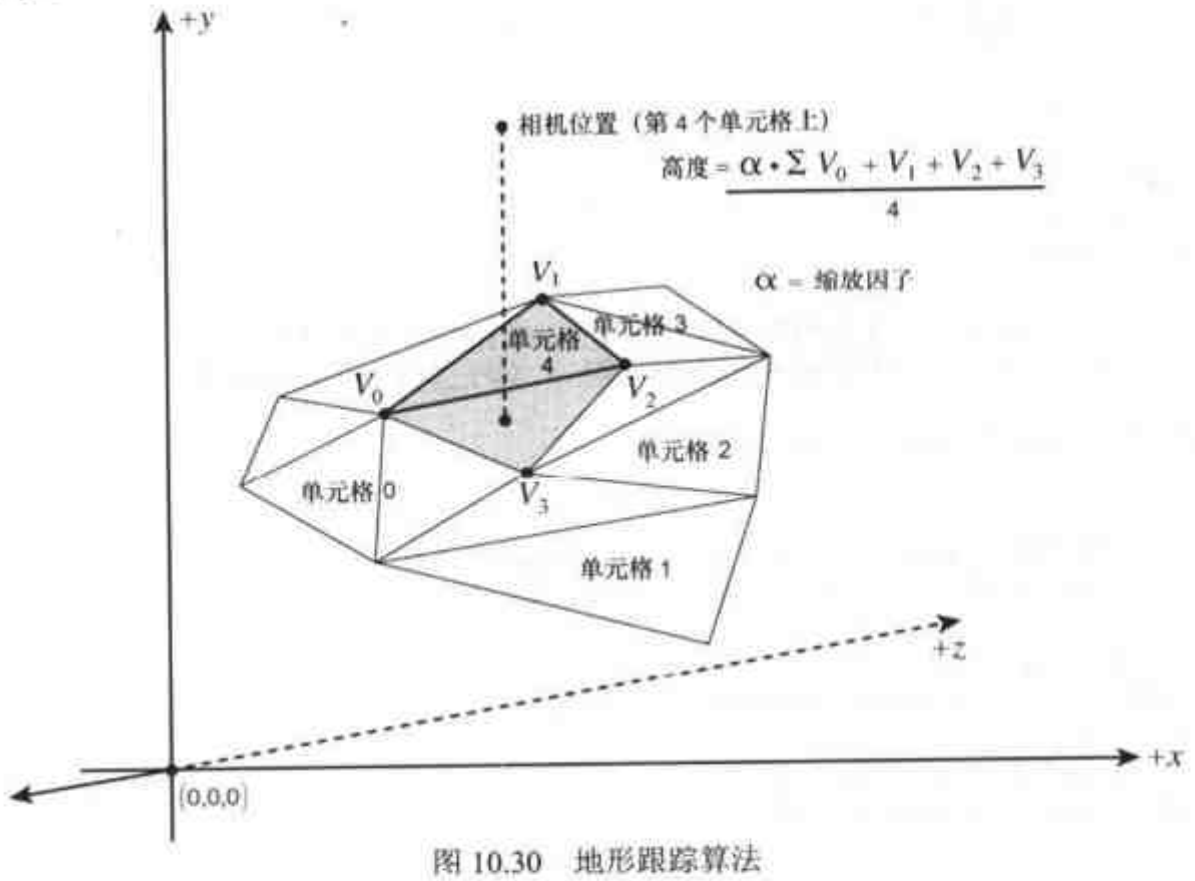


图 10.30　地形跟踪算法

解决这种问题的方法很多。可以为沙地汽车创建一个真正的考虑了动量的物理模型，但对于这样简单的问题，这过于复杂。也可以走另一个极端，根据相机所在单元格的最大高度上下移动相机，但效果不会太好。因此作者走了一条中间路线：使用一个基于速度、重力和瞬时加速度（impulse acceleration）的简化物理模型，其中瞬时加速度取决于相机的位置和方向，后面说到沙地汽车时，指的也是相机，反之亦然。这个物理模型的要点如下：

- 沙地汽车有前进速度和前进方向，可使用箭头键来控制它们（玩家按住前进方向键或后退方向键时，加速度为常量）。
- 有一个向下的重力，它始终作用于沙地汽车。因此，沙地汽车在沿 y 向下的速度随时间的增长而加大。
- 有一个向上的作用于沙地汽车的法向力。如果沙地汽车在海平面之下，这个力将其往上推。
- 在任何时候，都根据沙地汽车在世界坐标空间中的位置来确定它在哪个地形单元格之上，并计算该单元格 4 个顶点的平均高度。然后将其同沙地汽车的高度（减去地面间隙）进行比较，如果后者更小，则将一个与高度差成正比的瞬时加速度用于沙地汽车，以提高其向上的速度。同时，对沙地汽车的最大高度进行限制，这样，检测到坡度或高度变化时，沙地汽车会向上跳跃，避免穿入地面。
- 除了修改沙地汽车的高度以跟踪地形外，还根据沙地汽车经过的地形梯度修改沙地汽车/相机的倾斜角，以实现车头根据地形上仰、俯冲的效果。
- 最后，还有保持物理模型稳定性的代码：在位置或方向变化足够小时，对其不予考虑，以避免振荡。

下面是模拟代码中使用的一些常量和变量：

```
// 为地形定义的常量
#define TERRAIN_WIDTH     4000
#define TERRAIN_HEIGHT    4000
#define TERRAIN_SCALE      700
#define MAX_SPEED        20

// 用于物理模型的变量，修改这些变量的值可获得不同的效果
float gravity  = -.40;  // 重力
float vel_y    = 0;     // 相机/汽车沿 y 轴的速度
float cam_speed = 0;    // 相机/汽车的速度
float sea_level = 50;   // 海平面
float gclearance = 75;   // 相机和地面之间的空隙
float neutral_pitch = 10;  // 相机的仰角
```

下面是地形跟踪算法的代码：

```
// 运动部分///
```

```
// 地形跟踪算法确定当前所在的单元格，然后计算其 4 个顶点处的平均高度
// 并根据沙地汽车的当前和地形的平均高度，将沙地汽车往上推

// 一些用于地形跟踪的变量
//ivar1 = columns;
//ivar2 = rows;
//fvar1 = col_vstep;
//fvar2 = row_vstep;

int cell_x = (cam.pos.x + TERRAIN_WIDTH/2) / obj_terrain.fvar1;
int cell_y = (cam.pos.z + TERRAIN_HEIGHT/2) / obj_terrain.fvar1;

static float terrain_height, delta;

// 检查是否在地形上
if ( (cell_x >=0) && (cell_x < obj_terrain.ivar1) &&
   (cell_y >=0) && (cell_y < obj_terrain.ivar2) )
  {
  // 计算当前单元格顶点的索引
  int v0 = cell_x + cell_y*obj_terrain.ivar2;
  int v1 = v0 + 1;
  int v2 = v1 + obj_terrain.ivar2;
  int v3 = v0 + obj_terrain.ivar2;

  // 计算平均高度
  terrain_height = 0.25 * (obj_terrain.vlist_trans[v0].y +
              obj_terrain.vlist_trans[v1].y +
              obj_terrain.vlist_trans[v2].y +
              obj_terrain.vlist_trans[v3].y) ;

  // 计算高度差
  delta = terrain_height - (cam.pos.y - gclearance);
  // 检测是否穿入地面
  if (delta > 0)
   {
   // 给相机一个作用力
   vel_y+=(delta * (0.025));

   // 检测是否穿入地面，如果是，立刻向上移动
   // 以免穿透几何体
   cam.pos.y+=(delta*.3);

   // 根据前进速度和地形坡度调整仰角
   cam.dir.x -= (delta*.015);

   } // end if

  } // end if

// 减速
if (cam_speed > (0.25) ) cam_speed-=.25;
else
if (cam_speed < (-0.25) ) cam_speed+=.25;
else
  cam_speed = 0;

// 播放发动器声音
DSound_Set_Freq(car_sound_id,8000+fabs(cam_speed)*250);

// 将相机的方向限制在一定范围内
if (cam.dir.x > (neutral_pitch+0.3)) cam.dir.x -= (.3);
else
if (cam.dir.x < (neutral_pitch-0.3)) cam.dir.x += (.3);
 else
  cam.dir.x = neutral_pitch;
```

```
// 应用重力
vel_y+=gravity;

// 检查是否在海平面下，如果是，将其置于海平面上
if (cam.pos.y < sea_level)
  {
  vel_y = 0;
  cam.pos.y = sea_level;
  } // end if

// 移动相机
cam.pos.x += cam_speed*Fast_Sin(cam.dir.y);
cam.pos.z += cam_speed*Fast_Cos(cam.dir.y);
cam.pos.y += vel_y;
```

代码很短。系统的输入为相机当前的位置，仅此而已。系统根据这些信息进行运行。

这个小型演示程序就介绍到这里。读者可对其中的变量进行调整，以模拟从 100 万吨的坦克到水上艇筏的各种东西。另外，只需做少量的工作，就可以以物体的方式加入多个地形片，将该演示程序修改成 3D 赛车、赛艇、冲浪、滑雪等游戏。创建滑雪演示程序确实很简单，例如，可以创建 16 或 32 个地形片，每个包含 32×32 个分片，在世界坐标空间中的大小为 2000×2000，然后将它们拼贴起来，组成一个大型滑道。使用物体剔除函数来剔除位于远裁剪面外侧的地形片，使用裁剪函数来处理视野内的地形片，将滑雪板置于相机的前面，并根据地形高度修改相机高度，这样就成了。

10.5 总　　结

很高兴这一章又结束了。作者已经厌烦了将近裁剪面置于远离视平面的地方以避免负投影的做法；但现在引擎更健壮。此时读者对裁剪算法应该有深入了解，它们一点也不复杂，而只是涉及大量细节而已。另外，添加剔除特性后，可将完全位于视景体外但不是背面的多边形全部剔除，这确实可以提高速度，因此现在引擎的速度更快。最后，本章简要地介绍了地形，并创建了一个小型演示程序，读者只需做少量的工作，便可将其修改为一个水上游戏或越野游戏，当然它很粗糙，不能与 Splashdown 或 V-Rally 同日而语。最后但并非最无关紧要的是，以后将不再提供对 8 位模式的支持。

第 11 章　深度缓存和可见性

本章介绍最基本的隐藏面消除/可见性判断方法之一——z 缓存。和以前一样，本章将重新编写引擎以支持这种技术，并提供两个演示程序，包括以下主要主题：

- 深度缓存和可见性简介；
- 平面方程；
- z 坐标插值；
- 1/z 缓存；
- 创建 z 缓存系统；
- 在光栅化函数中加入 z 缓存支持；
- 可能的 z 缓存优化；
- 新软件和 z 缓存演示程序。

11.1　深度缓存和可见性简介

重要的内容先介绍。下列两个概念常常被混为一谈：

- 隐藏面消除；
- 可见面判定。

其原因是，在有些情况下，它们确实是一回事；但在其他情况下却是两码事。例如，在 3D 流水线中执行背面消除时，执行的就是隐藏面消除。在另一方面，裁剪既是隐藏面消除（将不在视景体内的区域裁剪掉），也是可见面判定——判断哪些多边形可见，无需进行裁剪。对于这两个概念之间有何差别这个问题，没有简单的答案。从现在开始，除非明确说消除，否则说到该做什么和不该绘制什么时，指的都是可见面判定（VSD）。很多图书也遵循这样的规则，读者对此不会感到惊讶。另外，讨论 z 缓存时，算法是在像素级进行的，不涉及到面，只涉及到面的组成部分。总之，使用哪个术语无关紧要，在容易发生混淆时，作者将做说明。

现在，读者应该能够得心应手地编写 3D 引擎，本书前面已经介绍过渲染和 3D 流水线的各个主要方面。当然，物理学和博弈（gameplay）不在本书的介绍范围内，本书的重点是变换、光照和多边形渲染。那么现在的情形是什么样的呢？已经有非常先进的光照系统，能够进行纹理映射以及在 2D 和 3D 空间进行裁剪，但多边形模型和环境还过于简单。其原因有两个。首先，由于没有合适的工具和文件格式，没有加载室内关卡（interior level），不过已经具备对其进行渲染的技术。第二个原因更微妙些：不能对多边形进行排序，即绝对准确地确定多边形的绘制顺序。

到目前为止，我们使用画家算法来对多边形进行排序，这种算法根据渲染列表中每个多边形的平均、最小或最大 z 值进行排序。然后，我们像画家那样按从后到前的顺序绘制多边形，在大多数情况下，结果都是正确的。然而，我们知道，这种算法有一些缺点，例如，它不能妥善地处理很长的多边形，即允许多边形相互贯通，如图 11.1 所示。

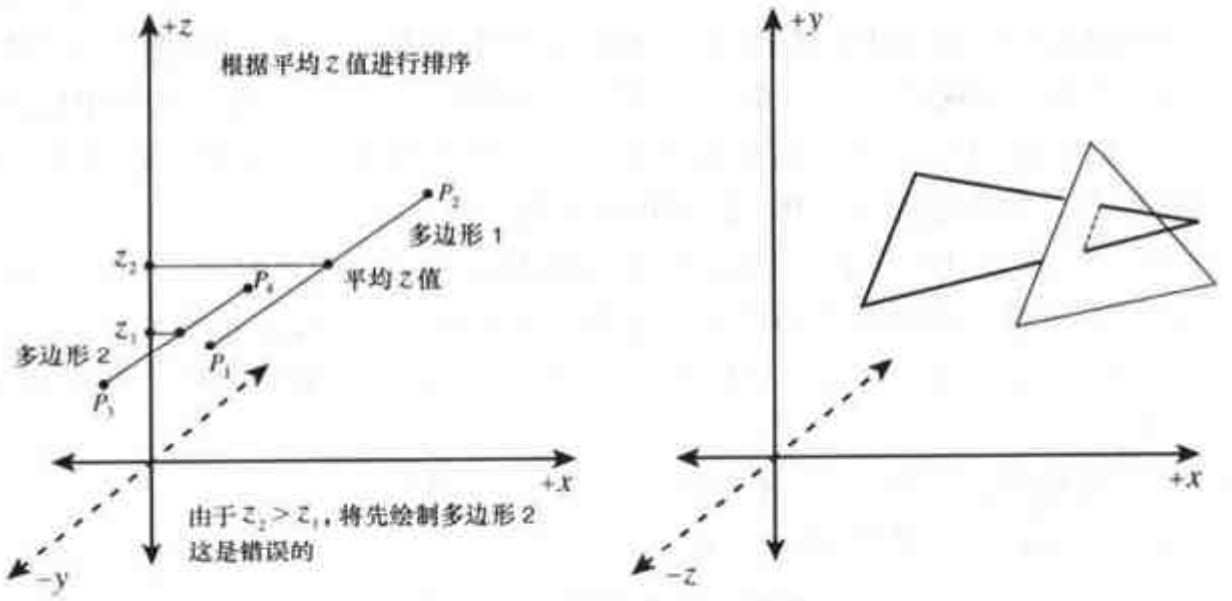


图 11.1　z 排序不管用的情形

好消息是，到目前为止，对于我们使用的 99%的多边形数据，画家算法都是可行的。对于那些主要由物体构成的场景而言，画家算法很管用，在物体不相互重叠且由小型多边形组成时尤其如此。另外，在渲染诸如地形等由规则且不重叠的几何体组成的网格时，这种算法也管用。地形虽然是 3D 的，但从很大程度上说，它只不过是高度变化的 2D 物体而已。

然而，在多边形相互贯通、包含大小各异的几何体时，画家算法（以下称之为 z 排序）可能不管用。我们需要另一种确定多边形渲染顺序的方法。初一想，我们可能决定重新编写 z 排序算法，使之更“聪明”。这实际上是不可能的。例如，可以每次考虑两个多边形，通过测试来确定它们的正确顺序。这类算法通常被称为列表优先顺序算法（list priority algorithm），它还有众多其他的名称，但它们都属于常识性的东西。概括地说，对于经过 z 排序的列表，为确保任何两个多边形的顺序都是正确的，需要做进一步测试。下面介绍 Newell-Sancha 算法使用的一组测试。

请看图 11.2 所示的两个多边形 P1 和 P2，根据最大 z 值对它们进行了 z 排序。因此在列表中，离视点较远的多边形位于前面。

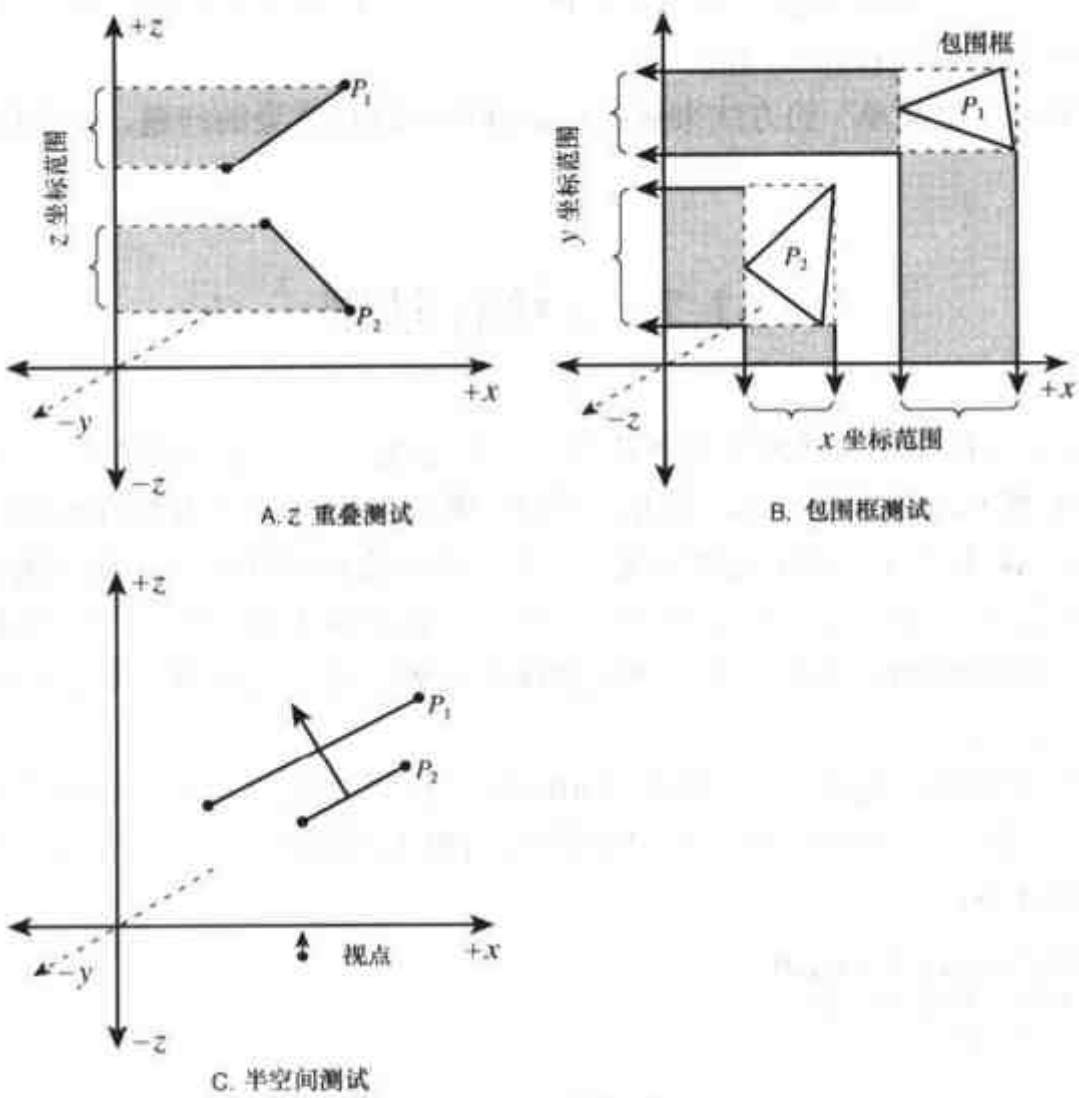


图 11.2　遮掩测试

● 测试 1（z-重叠测试）：如果 P1 和 P2 的 z 坐标范围不重叠，则 P1 不可能遮掩 P2。换句话说，如果 $P1_{min}>P2_{max}$，将 P1 绘制到帧缓存中，图 11.2a 说明了这种情形。另一方面，如果 $P1_{min}<P2_{max}$，则 P1 可能遮掩 P2，虽然在列表中 P1 排在前面，首先被绘制。在这种情况下，还需要进行下述测试，如果其中任何一个测试得到肯定回答，则算法结束，P1 不可能遮掩 P2。

● 测试 2（包围框测试）：P1 和 P2 的包围框在 x 轴和 y 轴方向上重叠吗？也就是说，如果 P1 和 P2 的 x 坐标和 y 坐标范围不重叠，则这两个多边形不可能彼此遮掩。图 11.2b 说明了这一点。

● 测试 3（半空间测试）：从视点看，P1 的半空间是否更远？如果是这样，则绘制 P1，结束算法。图 11.2c 说明了这一点。

● 测试 4（半空间测试）：从视点看，P2 的半空间是否更近？如果是这样，则绘制 P1，结束算法。

● 测试 5（透视测试）：P1 和 P2 的透视投影是否重叠？也就是说，对多边形进行投影后，它们的投影是否重叠？如果不重叠，则绘制 P1，结束算法。

如果上述测试都没有通过，则交换 P1 和 P2，并标记这一点。如果再次发生需要互换的情况，则表明存在循环遮掩，如图 11.3 所示。在这种情况下，必须将 P1 或 P2 进行分割。

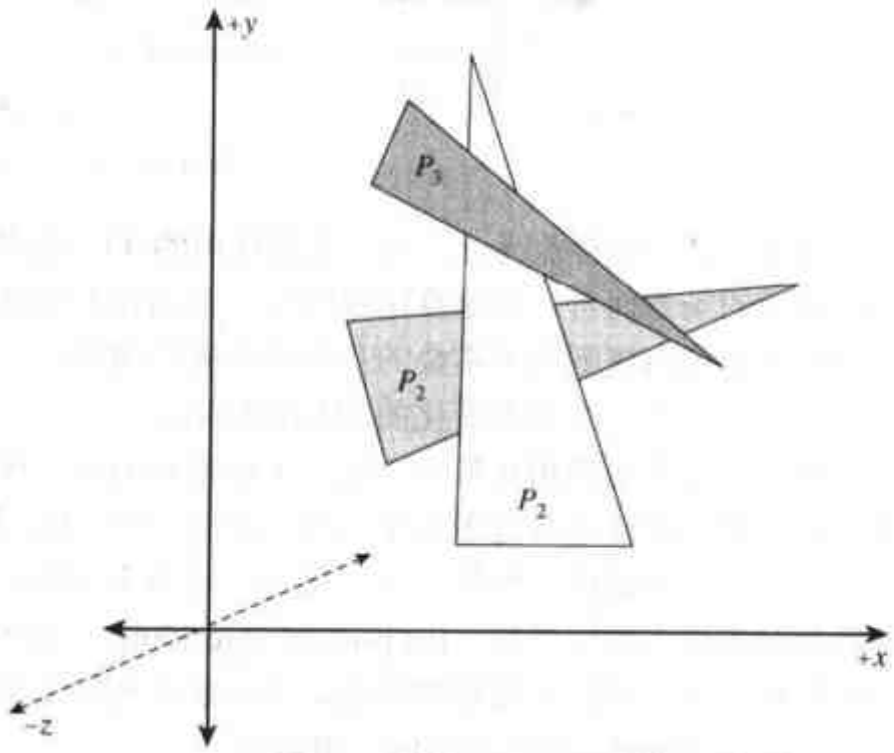


图 11.3 多边形循环遮掩

如果读者没有看懂上述算法，不用担心——没有人会使用它。这里介绍它旨在让读者了解整体流程。基本上，这种算法是不断提出越来越复杂的问题，直到确定正确的顺序为止。这样做值得吗？答案是否定的。如果要得到绝对正确的顺序，您不会使用这种首先带来灾难的算法；如果不想得到绝对正确的顺序，也不会使用这种算法，因为在这种情况下，z 排序是可行的。很多 PS 和 PC 游戏使用的都是 z 排序，而玩家并没有发现有什么问题。

我们需要一种更精致（更简单）的方法来解决按正确顺序进行渲染的问题，避免使用上述复杂的测试。这就是 z 缓存。

11.2 z 缓存基础

具有讽刺意味的是，z 排序为上述所有问题提供了解决方案。对于小型多边形，z 排序很管用，但随着多边形的大小差异越来越大，将出现异常，这是因为排列顺序不正确（如我们在前面分析算法的范例中看到的）。然而，如果基于像素进行 z 排序将如何呢？这样，将不会出现顺序不正确的情况，因为将对最小的屏幕单位（像素）进行排序。这就是 z 缓存的工作原理——它是像素级 z 排序，即像素级画家算法。

令人惊讶的是，这种思想如此简单，让人感觉就像是作弊。这种算法最初是由 Edwin Catmull 于 1974 年发明的，它看似非常简单：

对于大小为 $M\times N$ 的屏幕，创建一个 z 缓存 zbuffer[x, y]，其中包含每个多边形中每个像素被光栅化或扫描转换后的 z 值。z 缓存的大小也是 $M\times N$，被初始化为最大可能 z 值（实际上是无穷远）。

使用 z 缓存的算法如下：

```
for (渲染列表中的每个多边形) begin
   1. 对其进行光栅化，生成 xi、yi、zi
    for(每组 xi、yi、zi)

   2. if(zi < zbuffer[xi, yi]) then(将 zi 写入到 Z 缓存中，
```

```
        即 zbuffer[xi, yi] = zi，然后将该像素显示到屏幕上：Plot(xi, yi))
    end for
```

简单地说，创建一个整型或浮点数数组，用于记录每个多边形被扫描转换后的 z 值。然而，不是简单地将每个像素显示到屏幕上，而是对像素的 *z* 值同 *z* 缓存中相应位置的当前 z 值进行比较。如果在被扫描转换的多边形中，像素(xi，yi)的 *z* 值小于 zbuffer[*xi*，*yi*]的当前值，则使用新的 *z* 值 *zi* 更新 *z* 缓存中的值，并将该像素绘制到屏幕上；否则，什么也不做。图 11.4 说明了这种算法。

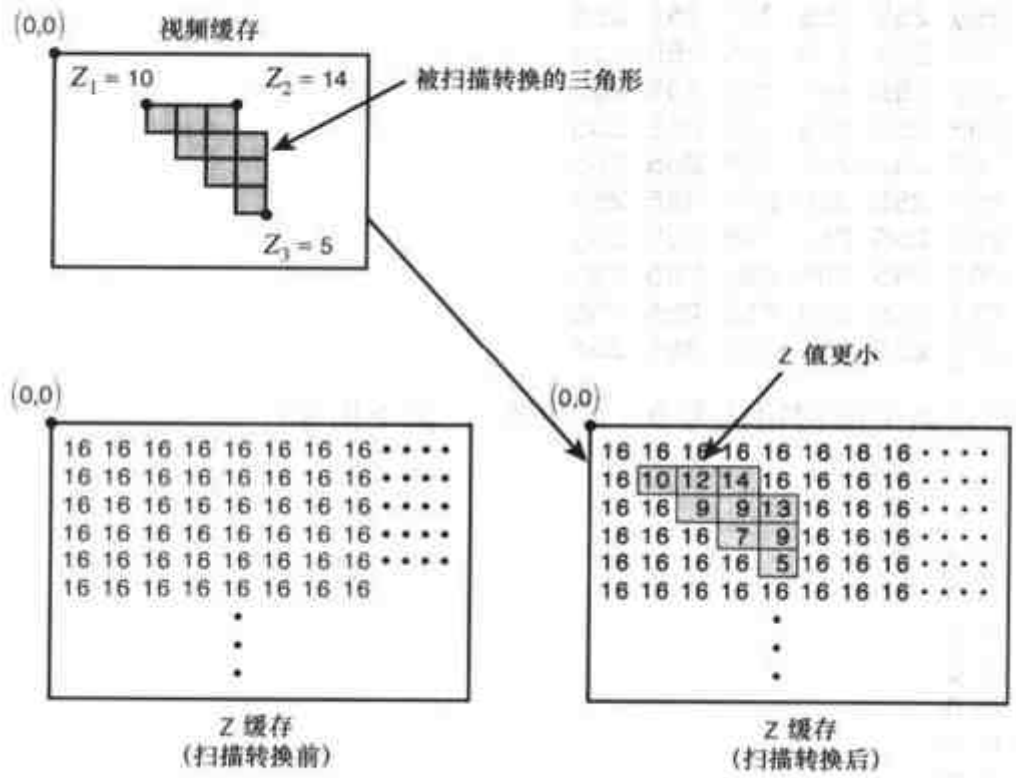


图 11.4　*z* 缓存算法

这种算法不断在更远的像素上绘制更近的像素，同时更新 2D *z* 缓存数组，基本上是一种像素级的画家算法。当然，有很多细节需要解决，后面将介绍它们，但在此之前先来看它存在一些显而易见的问题。

11.2.1　*z* 缓存存在的问题

首先，必须找到一种方法，用于计算被光栅化的多边形的 *z* 值。这可能比较困难，调用光栅化函数时，通常没有多边形的 *z* 信息。这个问题必须解决，另外还需要决定在哪个坐标空间内计算 *z* 值。其次，即使能够将 *z* 值传递到光栅化函数中，计算量也太大了，将把光栅化函数压垮，因为光照计算、纹理映射和裁剪已经使它够呛。最后，*z* 缓存占据的存储空间与屏幕缓存一样多，甚至更多，因此需要的内存量将增加一倍。

总之，使用 *z* 缓存后，将降低光栅化函数的速度，重复绘制像素，占用的内存量将至少增加一倍。看起来，*z* 缓存没有多大帮助。具有讽刺意义的是，要进行透视修正纹理映射，必须计算 *z* 值，因此无论如何以后都需要使用 *z* 缓存。至于重新绘制像素的问题，也有办法避免；但使用纯粹的 *z* 缓存时，重新绘制是不可避免的；因此最好的办法是提高其速度。最后，当前普通计算机至少有 128MB 的内存，而每个像素为 32 位时，800×600 的 *z* 缓存只占用 2MB 的内存，这没有什么大不了的。

11.2.2　*z* 缓存范例

实现 *z* 缓存算法之前，为确保读者理解它，先来看一个简单的例子。假设有一个 10×10 像素的 *z* 缓存(20 世纪 70 年代末的计算机就能处理了)，其中每个元素都是 8 位的整数，因此将对小数值进行四舍五入。可以使用下述数据结构来存储该 *z* 缓存：

```
UCHAR zbuffer[10][10];
```

或者：

```
UCHAR zbuffer[10*10];
```

使用哪种数据结构无关紧要，只要能够满足要求即可。在这个例子中，假设可以使用表示法 zbuffer[*xi*][*yi*] 来存取数组（其中，*xi* 和 *yi* 的取值范围为 0～9）。假设有一个 10×10 的屏幕，屏幕缓存也是一个数组，我们将其称为 screen_buffer[ ][ ]，其寻址方案与 *z* 缓存相同。

将 *z* 缓存中的元素初始化为一个很大的值，不会将这样的值写入到 *z* 缓存中，如下所示：

```
zbuffer[][] =

255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
```

并将 scree_buffer 的元素全部初始化为 0（即黑色），如下所示：

```
screen_buffer[][]=

0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

现在，绘制一个从(0，0)到(3，3)的矩形，其 *z* 值为 100，颜色为 5。由于 100 小于 255，因此在 *z* 缓存中，(0，0)到(3，3)的元素将被改为 100，同时屏幕缓存中相应区域内的像素颜色将为 5。现在，zbuffer[ ][ ] 和 screen_buffer[ ][ ]如下：

```
zbuffer[][] =

100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255

screen_buffer[][]=

5 5 5 5 0 0 0 0 0 0
5 5 5 5 0 0 0 0 0 0
5 5 5 5 0 0 0 0 0 0
5 5 5 5 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

注意：以粗体显示的是 zbuffer[ ][ ]和 screen_buffer[ ][ ]中被修改的元素。

接下来再绘制一个(2，2)到(5，4)的矩形，其 z 值为 150，颜色为 8。150 大于 100，但小于 255，因此现在的 zbuffer[ ][ ]和 screen_buffer[ ][ ]如下：

```
zbuffer[][] =

100 100 100 100 255 255 255 255 255 255
100 100 100 100 255 255 255 255 255 255
100 100 100 100 150 150 255 255 255 255
100 100 100 100 150 150 255 255 255 255
255 255 150 150 150 150 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255
255 255 255 255 255 255 255 255 255 255

screen_buffer[][]=

5 5 5 5 0 0 0 0 0 0
5 5 5 5 0 0 0 0 0 0
5 5 5 5 8 8 0 0 0 0
5 5 5 5 8 8 0 0 0 0
0 0 8 8 8 8 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

正如读者看到的，第一个矩形的 z 值小于第二个矩形的，因此重叠的区域不受影响。然而，zbuffer[ ][ ]和 screen_buffer[ ][ ]中被新矩形覆盖的其他区域被更新。最后，绘制一个从(0，0)的(9，9)的矩形，其 z 值为 50，颜色为 1。现在的 zbuffer[ ][ ]和 screen_buffer[ ][ ]如下：

```
zbuffer[][] =

50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50
50 50 50 50 50 50 50 50 50 50

screen_buffer[][]=

1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
```

从中可知，整个屏幕缓存和 z 缓存都被更新，在更新方面，这是最糟糕的情况。我们不希望这种情况大量发生，因为这样必须反复绘制像素。然而，与简单的 z 排序相比，z 缓存要好得多，使用 z 排序时，总是反复绘制像素，因为它总是绘制整个多边形，而不考虑其他因素。

注意：使用 z 缓存时需要对 z 值进行比较，但这样做得到的补偿是，如果新像素的 z 值更大，则无需

绘制它。在基于像素进行纹理映射和光照计算时，这有重大影响。

接下来介绍对多边形进行光栅化时如何计算 z 值。

11.2.3 平面方程法

对多边形进行扫描转换时，如何计算每个像素的 z 值呢？方法有几种，这里只介绍两种最简单的。第一种方法是，在扫描转换时使用平面的定义。我们知道，每个多边形的所有顶点都是共面的。在我们的系统中，所有多边形都必须满足这个要求，这也是三角形的物理特征。因此，对三角形进行扫描转换时，可以先计算该三角形所在平面的平面方程。为此，可以计算三角形的法线，得到点-法线形式的平面方程得，如下所示：

```
nx*(x-x0) + ny*(y-y0) + nz*(z-z0) = 0
```

其中<nx，ny，nz>是多边形的法线，(x0，y0，z0)是多边形上的一个点。
然后将上述方程转换为下述形式：

```
a*x+b*y+c*z + d = 0
```

其中 a = nx，b = ny，z = nz，d = -（nx*x0+ny*y0+nz*z0)。
然后求解 z，结果如下：

```
z = -(a*x + b*y + d)/c
```

这样便可以计算 z 值，因为扫描转换函数将生成 x 和 y。这种方法的惟一缺点是，必须计算平面方程和其中的系数，同时，为计算每个像素的 z 值，需要执行两次乘法、一次除法和两次加法运算。当然，可以减少这种计算量。首先，可以使用多边形中预先计算好的面法线；其次，可以使用前向差分和增量计算来简化 z 值的计算，就像 Gouraud 着色和标准三角形扫描转换函数中那样。我们发现，每前进一个像素，x 的值增加 1.0。因此，可以推导出增量计算公式，接下来将对此进行详细介绍。

1. 沿 x 轴方向移动时 z 值的变化情况

我们注意到：

$$z_i = -(a*x + b*y + d)/c$$

移动到当前扫描线的下一个像素时：

$$\begin{aligned} z_{i+1} &= -(a*(x+1)+b*y + d)/c \\ &= -(a*x + a + b*y + d)/c \\ &= -(a*x + b*y + d)/c + (-a/c) \\ &= z_i + (-a/c) \end{aligned}$$

因此，水平移动时，下一个 z 值为前一个 z 值加上常数项-a/c。因此对于每个像素，z 坐标插值被简化为一次加法运算。当然，仍然需要计算每条扫描线上第一个像素的 z 值，但这也可以通过增量算法来完成，因为每向下移动一条扫描线，y 值增加 1.0。

2. 沿 y 轴方向移动时 z 值的变化情况

我们注意到：

$$z_i = -(a*x + b*y + d)/c$$

移动到下一条扫描线时：

$$\begin{aligned} z_{i+1} &= -(a*x+b*(y+1) + d)/c \\ &= -(a*x + b*y + b + d)/c \\ &= -(a*x + B*y + d)/c + (-b/c) \\ &= z_i + (-b/c) \end{aligned}$$

这表明，向下移动一条扫描线时，下一个 z 值为前一个 z 值加上常数项-b/c。图 11.5 说明了对多边形进行扫描转换时，z 值沿 x 轴和 y 轴的变换情况。

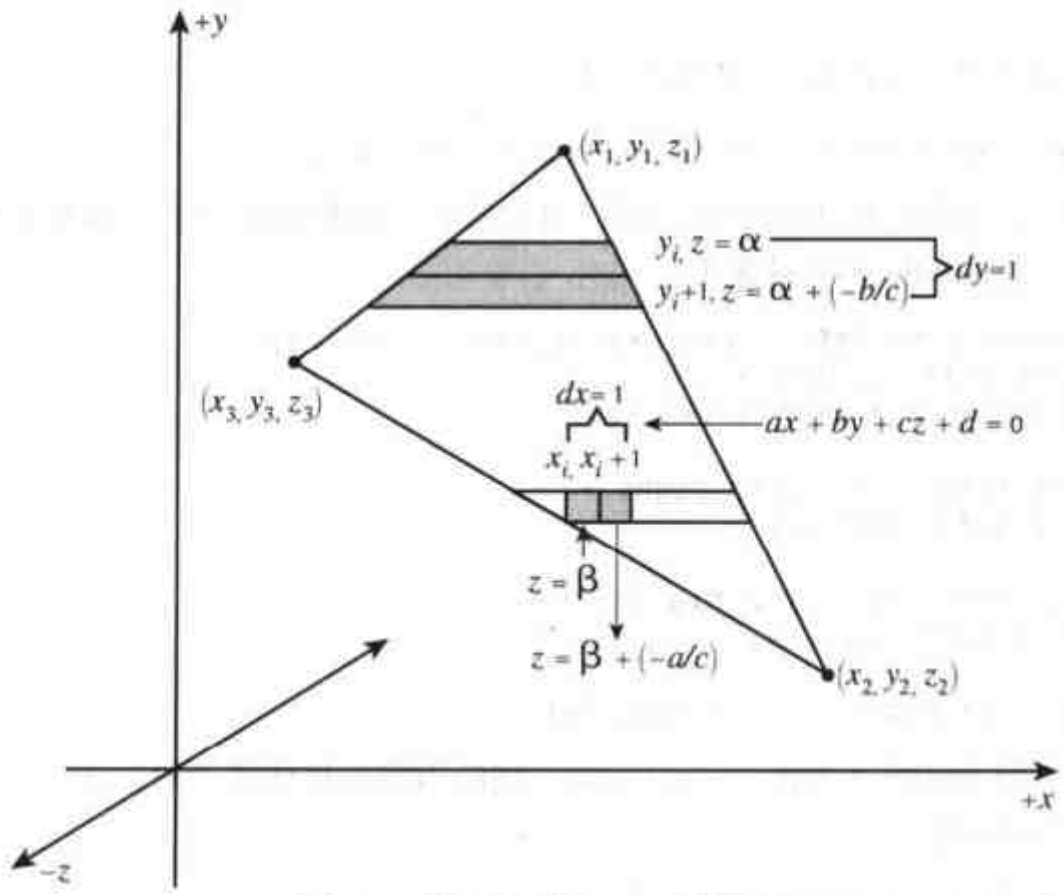


图 11.5　基于共面的增量计算公式

虽然使用平面方程来直接计算多边形的 z 值是正确的，但有两个原因使得不能这样做：它只在物体空间中是正确的（后面将再次讨论这一点）；可以像以前那样通过插值来计算 z 值。进行 Gouraud 着色时，根据三角形 3 个顶点的光照强度通过插值来计算每个像素的光照强度；进行仿射纹理映射时，通过插值来计算纹理坐标；可以使用同样的方法来计算 z 值。

11.2.4　z 坐标插值

图 11.6 说明了如何通过插值来计算 z 值，其中的三角形是平顶的，对于平底三角形，计算方法与此类似。因此，这里只推导平顶三角形的计算公式，但在最后的软件实现中将处理这两种情况。

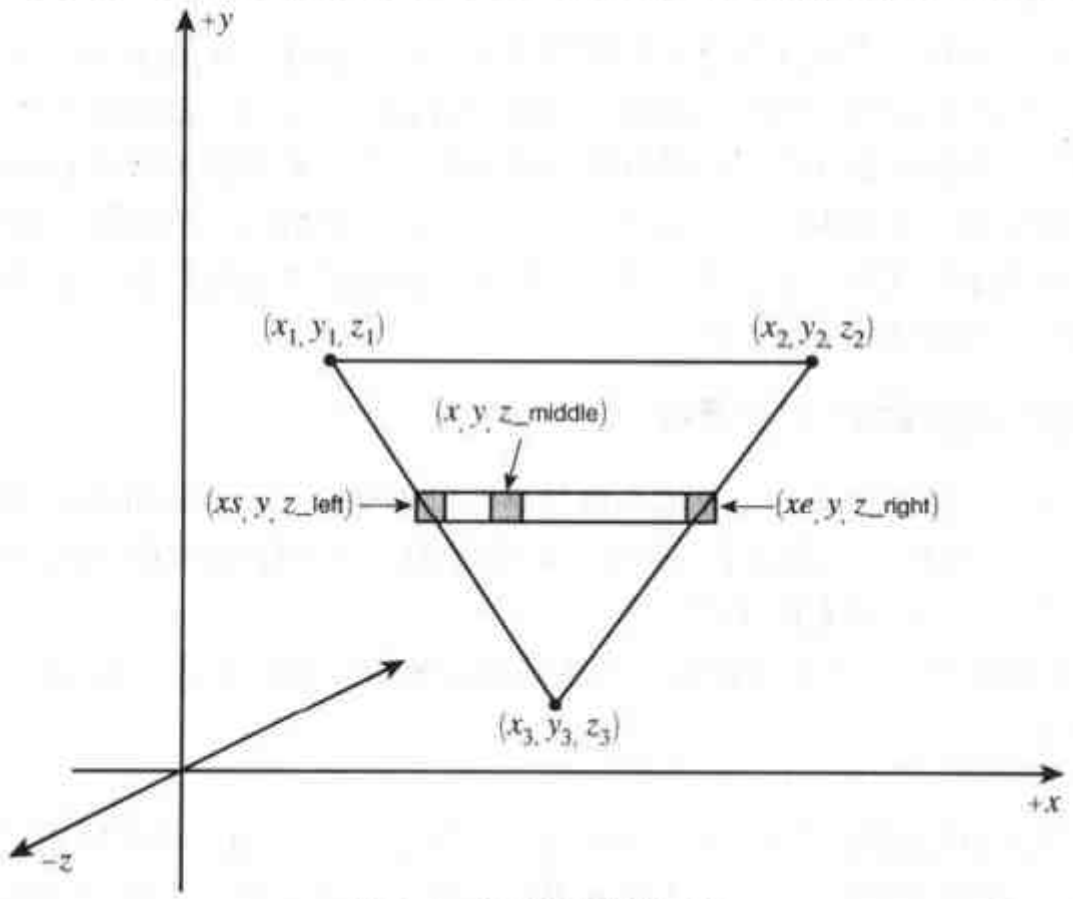


图 11.6　通过插值计算 z 值

通过插值计算 z_left、z_right 和 z_middle 的公式如下：

```
z_left = ((y3-y)*z1 + (y-y1)*z3)/(y3-y1);
```

```
z_right = ((y3-y)*z2 + (y-y1)*z3)/(y3-y1);

z_middle = ((xe - x)*z_left + (x-xs)*z_right)/(xe-xs);
```

这些公式有点复杂，因为它们不是增量式的。可以很容易地根据这些公式推导出增量版本。首先，推导沿 x 轴方向移动时，z_middle 的增量版本：

$$
\begin{aligned}
z\_middle_{i+1} &= ((xe-x-1)*z\_left + (x+1-xs)*z\_right)/(xe-xs) \\
&= ((xe-x)*z\_left - z\_left + \\
&\quad (x-xs)*z\_right + z\_right)/(xe-xs) \\
&= ((xe-x)*z\_left + (x-xs)*z\_right + \\
&\quad z\_right-z\_left)/(xe-xs) \\
&= ((xe-x)*z\_left + (x-xs)*z\_right)/(xe-xs) + \\
&\quad (z\_right-z\_left)/(xe-xs) \\
&= z\_middle_i + (z\_right-z\_left)/(xe-xs)
\end{aligned}
$$

如果令 delta_zx = (z_right − z_left) / (xe − xs)，则沿当前扫描线移动时，下一个像素的 z 值为前一个 z 值加上 delta_zx。换句话说：

$$z\_middle_{i+1} = z\_middle_i + delta\_zx$$

我们再次找到了一种算法，只需执行一次加法运算就可以得到下一个 z 值。然而，对于每条扫描线，还需要计算 z_right 和 z_left。同样，鉴于每向下移动一条扫描线，y 值增加 1.0，因此也可以使用前向差分推导出一种增量算法。这里省略推导过程，直接给出结果：

$$
\begin{aligned}
z\_left_{i+1} &= z\_left_i + (z3-z1)/(y3-y1) \\
z\_right_{i+1} &= z\_right_i + (z3-z2)/(y3-y2)
\end{aligned}
$$

如果令 delta_zyl = ($z3 - z1$) / ($y3 - y1$)，delta_zyr = ($z3 - z2$) / ($y3 - y2$)，则计算 z_left 和 z_right 的增量公式可简化为：

$$
\begin{aligned}
z\_left_{i+1} &= z\_left_i + delta\_zyl \\
z\_right_{i+1} &= z\_right_i + delta\_zyr
\end{aligned}
$$

上述公式表明，只需执行一次加法运算就能够得到下一条扫描线的 z_right 和 z_left。

有了插值公式后，只需在新的三角形扫描转换函数使用它们，并对 z 缓存进行处理即可。在此之前，作者要指出两点。首先，前面的推导使用的是标准线性插值，与纹理映射函数和 Gouraud Shader 中所做的完全相同。这两个函数都在三角形内部进行线性插值，因此，只需增加一个要通过插值计算的值——z 值。因此，如果要从空白开始编写代码，上述推导将很有帮助，但没有必要这样做，因为我们已经有类似的代码。其次，前面推导的 z 缓存算法是错误的。

11.2.5 z 缓存中的问题和 1/Z 缓存

当前的 z 缓存算法之所以是错误的，是因为推导是在世界坐标空间或相机坐标空间中进行的。我们使用的是三角形的(x，y，z)坐标，而忘记了三角形已经被投影，其坐标为屏幕空间的投影坐标(x'，y')。这是一个大问题。为明白这一点，请看图 11.7。

在 3D 空间中通过插值计算 z 值毫无用处，因为我们需要根据投影公式绘制 2D 三角形：

```
x_screen = d*x/z
y_screen = d*y/z
```

从图 11.7 可知，我们通过插值计算的是三角形在 3D 空间中的 z 值，但使用的却是投影后的 2D 坐标(x_screen 和 y_screen)。在屏幕空间中，z 值不是线性的，因此存在 z 缓存扭曲（artifacting），就像 Gouraud 着色和仿射纹理映射中一样。然而，在 Gouraud 着色和仿射纹理映射中，情况没有这么糟糕，只要多边形足够小且平行于视平面，扭曲是可以容忍的。使用 z 缓存时，根据投影后的（x，y）坐标来计算三角形在 3D 空间中的 z 值有同样的影响，计算得到的 z 值是错误的。如图 11.7 所示，从多边形到投影点的各条射线

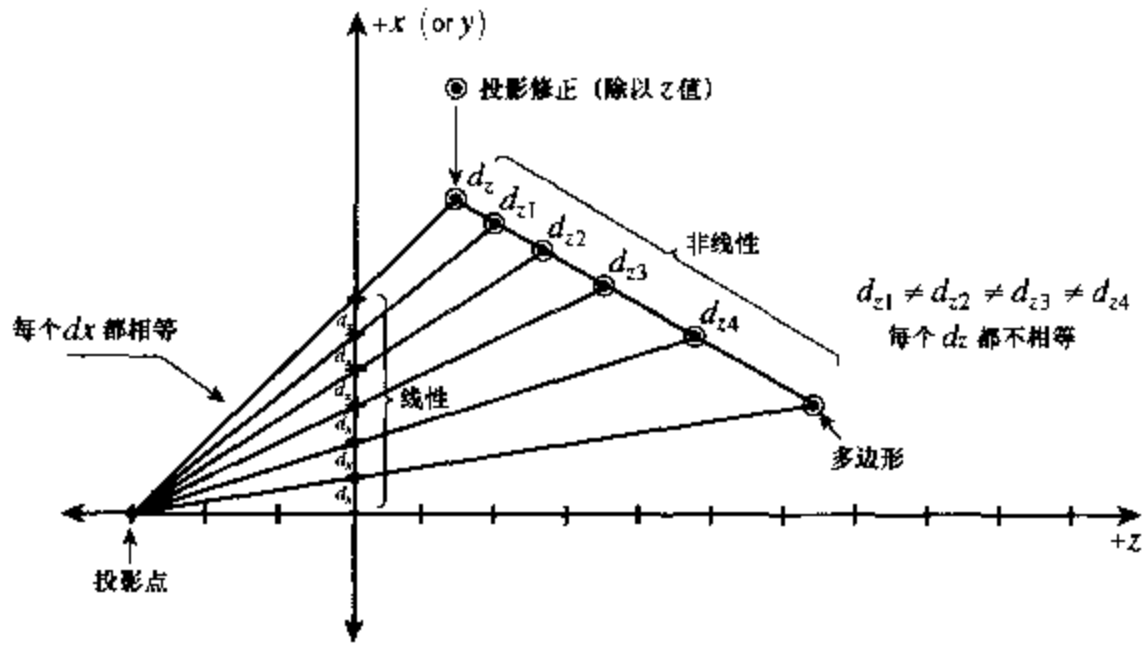


图 11.7　在屏幕空间中通过线性插值计算得到的 z 值是错误的

与视平面的交点沿 x 轴方向是均匀分布的，但这些射线与多边形边的交点并不是均匀分布的。事实上，这些点是按 1/z 均匀分布的。本书后面实现透视修正纹理映射时，将详细介绍这方面的数学计算，现在读者只需要知道在屏幕屏间中，z 不是线性变化的，但 1/z 是线性变化的即可。

那么，为何在多边形内部通过插值计算 z 呢？为何不通过插值计算 1/z（因为它是线性变化的）？这是一个好主意——后面将这样做。就现在而言，我们将通过线性插值来计算 z，不考虑其扭曲效果，因为事实证明 z 缓存的效果不错。然而，后面进行透视修正纹理映射时，将使用 1/z 缓存，并根据 1/z 的值来计算纹理映射坐标，因此是一举两得。现在，出于简化的目的，使用 z 缓存。

11.2.6　一个通过插值计算 z 和 1/z 的例子

介绍坚持使用标准 z 缓存的原因以及 1/z 缓存的优点后，下面通过一个例子让读者对其有全面的认识。John Carmack 说过，可向任何人提供的一条建议是：要有全面认识，必须深入领会基本原理。

这也是作者一贯坚持的理念。为达此目的，作者进行记录、使用图形和表格，进行检测和编码等，以确保最终理解它。要有“我现在明白了”的感觉，必须经过这些过程；否则，即使对工作原理有了解，也是不全面的。下面通过一个简单的例子来说明 z 缓存和 1/z 缓存。

图 11.8 说明了典型的透视投影。其中垂直轴为 y 轴，水平轴为 z 轴，x 轴垂直于纸面向上。视距为 2；处于简化的目的，我们假设多边形上所有点的 x 坐标都为 0，即多边形位于平面 x=0 上。

有一个需要对其进行投影、光栅化和 z 缓存的三角形。要处理的第一条边是 **p0**（0，12，4）->**p1**（0，5，7）。我们将计算屏幕投影点 **p0'**和 **p1'**，然后根据它们通过插值来计算投影点 **pa'**和 **pb'**，但它们的 z 值却是 pa*和 pb*的，这是错误的。首先，列出已知的信息：

p0 = (0,12,4), **p1** = (0,5,7)

使用下述公式对它们进行透视投影：

y' = (d/(d+z))*y

注意，对于 x 也需要执行相同的计算，但在这个例子中，我们假设 x = 0。

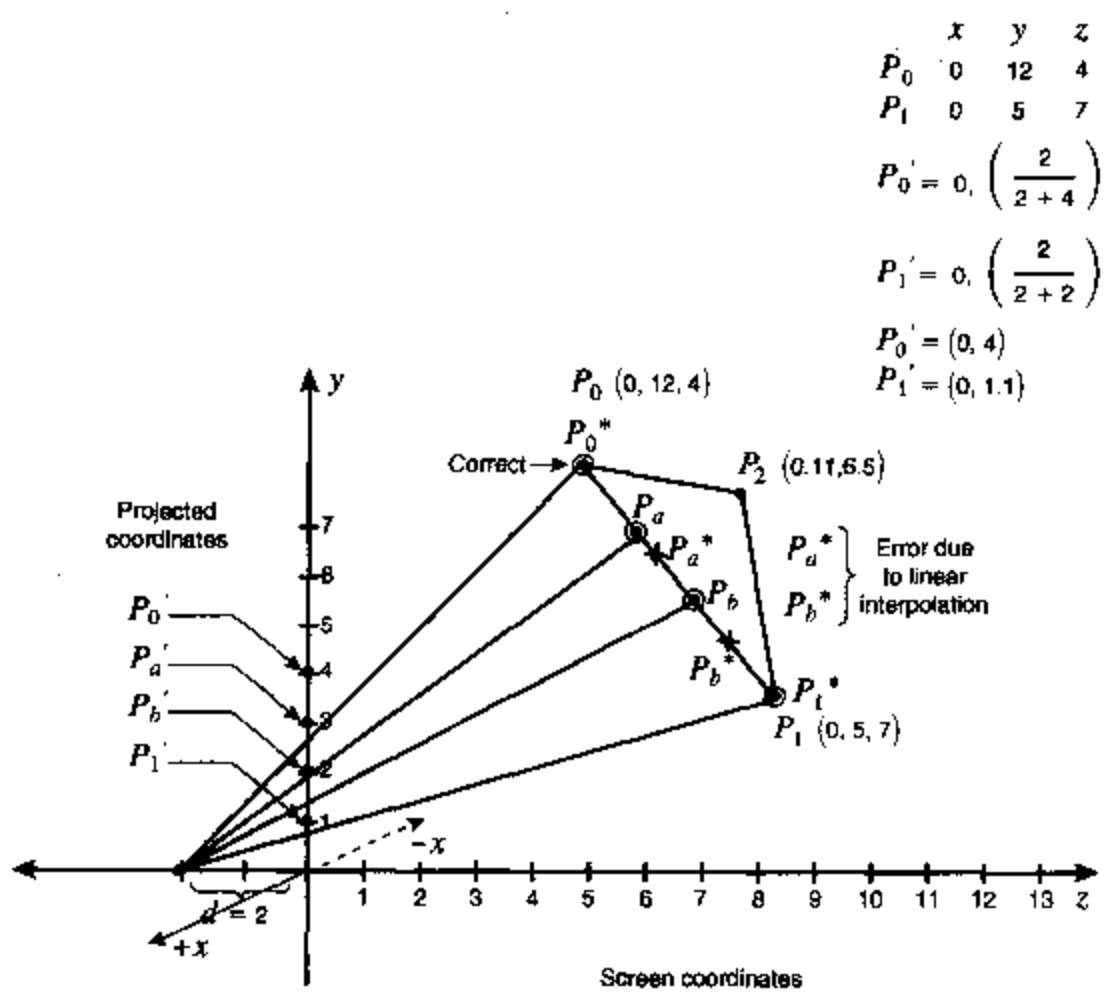
p0' = (0,(2/(2+4))*12,0) = (0, 4, 0)
p1' = (0,(2/(2+7))*5, 0) = (0, 1.111,0)

我们将根据投影后的坐标通过线性插值来计算 z 值，因此将 **p0'**和 **p1'**的 z 值设置为投影前的 z 坐标：

p0' = (0, 4, 4)
p1' = (0, 1.111, 7)

现在可以进行插值了，首先计算 dy 和 dz：

dy = p1'(y) - p0'(y) = (1.11 - 4.0) = -2.89
dz = p1'(z) - p0'(z) = (7 - 4) = 3



$$(x_s, y_s) \longrightarrow y_s = \frac{D}{(D+z)} \cdot y$$

$$x_s = \frac{D}{(D+z)} \cdot x$$

图 11.8　透视投影和 z 坐标插值误差分析

要在屏幕空间中绘制一条从 **p0'**到 **p1'**的直线，只需从 **p0'**开始，每次绘制一个像素，并将 y 值减 1.0，直到到达 **p1'**为止。y 值每增加 1，z 值增加 dz/dy：

```
dz/dy = (3)/(-2.89) = -1.03
```

我们从屏幕空间中的 **p0'**(0，4)开始，然后绘制 3 个像素。表 11.1 列出了每一步的情况。

表 11.1　　z 插值范例 1

迭代	x	y	z
[0]开始	$x = 0$	$y = 4$	$z = 4$
[1] dy = -1, dz/dy = -1.03	$x = 0$	$y = 3$	$z = 4+1.03 = 5.03$
[2] dy = -1, dz/dy = -1.03	$x = 0$	$y = 2$	$z = 5.03+1.03 = 6.06$
[3] dy = -1, dz/dy = -1.03	$x = 0$	$y = 1$	$z = 6.06+1.03 = 7.09$

从图 11.8 可知，在屏幕空间进行插值时，在正确的情况下，计算得到的 **pa'**和 **pb'**的 z 值应分别是 **pa** 和 **pb** 的 z 值；但通过线性插值得到的分别是 **pa***和 **pb***的 z 值。这种误差是由于线性插值引起的。多边形边越接近于平行视平面，这种误差越小；反之越大。

从下述的角度很容易理解问题所在：世界坐标到屏幕坐标的变换并非线性的（由于变换时除以了 z），每当 y 值变化一个单位时给 z 值加上一个常量（这里为-1.03）能得到正确结果吗？不能。我们需要基于在屏幕空间呈线性变化的东西进行采样，如 1/z，这样投影和插值的影响将相互抵消，使样本呈线性变化。

至此，读者应明白以线性插值来计算 z 值是错误的。如果读者打开 10 本 3D 图形学方面的图书，将发

现它们在这方面的内容都是错误的，更准确地说，没有详细讨论这方面的细节。接下来通过线性插值来计算 1/z。和前面一样，从投影点开始：

```
p0' = (0, 4, 4)
p1' = (0, 1.11, 7)
```

但将最后一个坐标改为 1/z：

```
p0' = (0,4,0.25)
p1' = (0,1.11,.142)
```

现在可以开始插值了。首先计算 dy 和 dz（这里的 z 实际上是 1/z）：

```
dy = p1'(y) - p0'(y) = (1.11 - 4.0) = -2.89
dz = p1'(z) - p0'(z) = (1/7 - 1/4) = -.107
```

同样，要在屏幕空间中绘制一条从 **p0'**到 **p1'**的直线，只需从 **p0'**开始，每次绘制一个像素，并将 y 值减 1.0，直到到达 **p1'**为止。y 值每增加 1，z 值增加 dz/dy：

```
dz/dy = (-.107)/(-2.89) = 0.037
```

我们从屏幕空间中的 **p0'**（0，4）开始，然后绘制 3 个像素。表 11.2 列出了每一步的情况。

表 11.2　　z 插值范例 2

迭代	x	y	z
[0] 开始	x = 0	y = 4	z = 1/4 = 0.25
[1] dy = -1，dz/dy = -0.037	x = 0	y = 3	z = 0.25-.037 = 0.213
[2] dy = -1，dz/dy = -0.037	x = 0	y = 2	z = .213-.037 = 0.176
[3] dy = -1，dz/dy = -0.037	x = 0	y = 1	z = 0.176+.037 = 0.139

令人惊讶的是，上述结果是正确的。

z 缓存很好理解，但如果通过比较 1/z 来更新 1/z 缓存呢？只需将比较方向反转，即如果像素的 1/z 大于 1/z 缓存中对应的值，则更新 1/z 缓存，并绘制该像素。

11.3　创建 z 缓存系统

有关理论就介绍到这里——作者快被这些小数字弄疯了。接下来实现 z 缓存。我们需要为 z 缓存分配内存，定义 z 缓存的大小，还可能需要一些定义 z 缓存的标记或属性。下面是作者创建的第 1 版 z 缓存数据结构：

```
// 为 z 缓存数据结构定义的常量
#define ZBUFFER_ATTR_16BIT   16
#define ZBUFFER_ATTR_32BIT   32

// z 缓存数据结构
typedef struct ZBUFFERV1_TYP
{
int attr;          // z 缓存的属性
UCHAR *zbuffer; // 指向存储空间的指针
int width;       // 宽度，单位为像素
int height;      // 高度，单位为像素
int sizeq;       // 大小，单位为四元组
} ZBUFFERV1, *ZBUFFERV1_PTR;
```

这个数据结构包含指向 z 缓存的指针、z 缓存的大小（有两个这样的变量，它们的单位分别是“字”和四元组）。字段 height 和 width 表示 z 缓存的高度和宽度，而 sizeq 是实际的四元组数目（每个四元组为 4 个字节）。另外，还有一个属性字段 attr，当前它只记录 z 缓存是 16 位还是 32 位的。就现在而言，我们将

使用 32 位的 z 缓存，因此并不需要 attr；但后面进行优化时，可能需要用到它。总之，我们只需要能够对 z 缓存执行下述操作：

- 创建 z 缓存；
- 删除 z 缓存；
- 使用一个值清空/填充 z 缓存（速度要快）。

下面是创建 z 缓存的函数：

```
int Create_Zbuffer(ZBUFFERV1_PTR zb, // 指向 z 缓存结构的指针
          int width,   // 宽度
          int height,  // 高度
          int attr)    // 属性
{
// 这个函数根据指定的参数创建一个 ZBUFFERV1

// 指针 zb 是否有效
if (!zb)
  return(0);

// 是否已经给 zb 指向的 ZBUFFERV1 分配了内存
if (zb->zbuffer)
  free(zb->zbuffer);

// 设置字段
zb->width  = width;
zb->height = height;
zb->attr   = attr;

// z 缓存是 16 位的还是 32 位的
if (attr & ZBUFFER_ATTR_16BIT)
  {
  // 计算四元组数
  zb->sizeq = width*height/2;

  // 分配内存
  if ((zb->zbuffer = (UCHAR *)malloc(width * height * sizeof(SHORT))))
   return(1);
  else
   return(0);

  } // end if
else
if (attr & ZBUFFER_ATTR_32BIT)
  {
  // 计算四元组数
  zb->sizeq = width*height;

  // 分配内存
  if ((zb->zbuffer = (UCHAR *)malloc(width * height * sizeof(INT))))
   return(1);
  else
   return(0);
  } // end if
else
  return(0);

} // end Create_Zbuffer
```

这个函数接受 ZBUFFERV1 指针（它指向要创建的 z 缓存）以及宽度、高度和属性。例如，要创建一个 800×600 的 32 位 z 缓存，可以这样调用它：

```
ZBUFFERV1 zbuffer; // storage for our Z buffer

Create_Zbuffer(&zbuffer, 800, 600, ZBUFFER_ATTR_32BIT);
```

ZBUFFERV1 中 z-buffer 是 UCHAR 类型的，在算法中可以将其转换为任何类型，如 int、float 等。在

我们将编写的光栅化函数中，将把它转换为无符号整型。

接下来的函数使用一个值来清空/填充 z 缓存。在每一帧中清除帧缓存时都必须这样做。通常，使用最大可能值或无穷大来填充 z 缓存。该函数如下：

```
void Clear_Zbuffer(ZBUFFERV1_PTR zb, UINT data)
{
// 这个函数使用一个值来清除/设置 z 缓存
// 填充内存时，每次总是填充一个四元组
// 因此，如果 z 缓存是 16 位的，必须使用用来填充的 16 位值的两个拷贝创建一个四元组
// 如果 z 缓存是 32 位的，只需将用于填充的值转换为 UINT 即可

Mem_Set_QUAD((void *)zb->zbuffer, data, zb->sizeq);

} // end Clear_Zbuffer
```

该函数调用了函数 Mem_Set_QUAD()，它位于 T3DLIB1.H 中，是一个内联的汇编语言函数，代码如下：

```
inline void Mem_Set_QUAD(void *dest, UINT data, int count)
{
// 这个函数以每次写入一个 32 位 UINT 值的方式填充/设置内存
// count 为四元组数

_asm
  {
  mov edi, dest  ; edi 指向目标内存
  mov ecx, count ; 要移动的 32 位字数
  mov eax, data  ; 32 位数据
  rep stosd    ; 移动数据
  } // end asm

} // end Mem_Set_QUAD
```

为最大限度地提高清空 z 缓存的速度，这里使用了汇编语言（实际上，也可以使用 SIMD 指令）。调用该函数来清空 z 缓存时，必须将填充数据转换为 UINT。另外，如果 z 缓存是 16 位的，必须创建两个包含最大值的字，并将它们拼接起来，如图 11.9 所示。

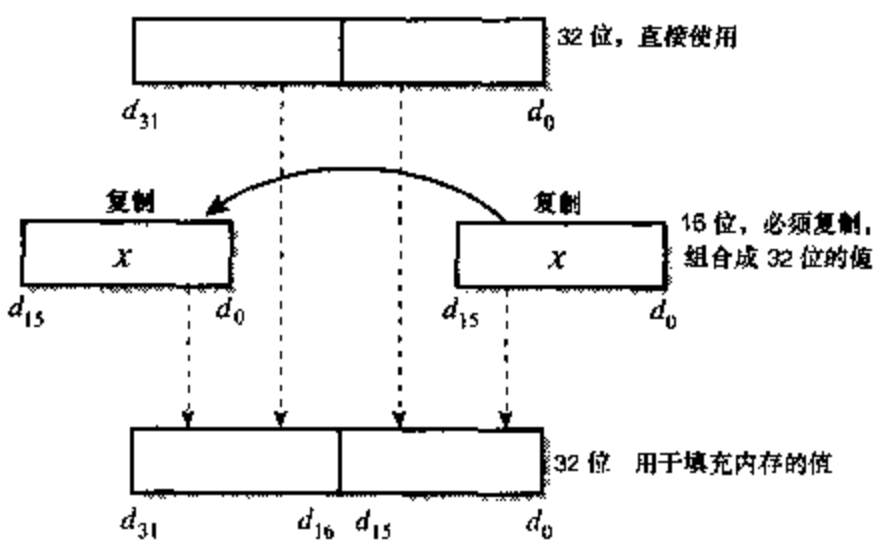


图 11.9　为进行 32 位高速 z 缓存填充做准备

例如，使用 16 位 z 缓存时，不能传递最大可能值的 32 位版本，而必须根据最大可能值创建一个 32 位值，使算法能正确地填充相邻的内存单元：

```
USHORT zmax16 = 10000; // say your max is 10000

unsigned int zdata =
 ( ( (unsigned int)zmax16 ) << 16 | (unsigned int)zmax16);
```

下面的例子演示了如何使用 32 位定点数 32 000 来填充 32 位的 z 缓存：

```
Clear_Zbuffer(&zbuffer, (32000 << FIXP16_SHIFT));
```

最后，要删除 z 缓存并释放它占用的内存，可以使用下面的函数：

```
int Delete_Zbuffer(ZBUFFERV1_PTR zb)
{
// 该函数删除 z 缓存并释放它占用的内存

// 检查指针 zb 是否有效
if (zb)
  {
  // 释放内存
  if (zb->zbuffer)
   free(zb->zbuffer);
```

```
    // 清除内存中的数据
    memset((void *)zb,0, sizeof(ZBUFFERV1));

    return(1);

    } // end if
  else
    return(0);

  } // end Delete_Zbuffer
```

要删除 z 缓存，只需使用指向它的指针来调用函数 Delete_ZBuffer()，该函数将释放 z 缓存占用的内存，并将 z 缓存清除。

提示：通过将 z 缓存抽象化为 ZBUFFERV1 类型，可以提供多个 z 缓存，实现一些特殊效果。

在光栅化函数中加入对 z 缓存的支持

要在光栅化函数中加入对 z 缓存的支持，首先需要确定哪些光栅化函数需要这种支持。当前，我们有下列 4 种光栅化函数：

- 第 1 类：支持固定/恒定着色；
- 第 2 类：支持 Gouraud 着色；
- 第 3 类：支持纹理映射和固定着色；
- 第 4 类：支持纹理映射和恒定着色。

第 1 类光栅化函数是最老的，它们来自《Windows 游戏编程大师技巧》，混合使用整数和浮点数，而不是定点数。这并不是说它们的速度更慢或更快，只是说它们不同于第 2～4 类光栅化函数，后者基于同一个核心光栅化函数/插值函数。一种情况是插值计算颜色（Gouraud Shader），另一种情况是插值计算纹理坐标（纹理映射函数）。首先要做出的决定是，需要对哪些光栅化函数进行升级。作者将对所有光栅化函数进行升级。然而，对于第 1 类光栅化函数，必须重新编写，因为它们没有插值功能，在原来的基础上加入 z 缓存支持很困难。

令人欣慰的是，对其他函数进行升级只需要增加大约 10 行代码。也许更多些，但不会太多。这是因为它们已经包含对多个变量进行插值的代码——在 Gouraud Shader 中，是对 RGB 颜色进行插值；在纹理映射函数中，是对纹理坐标进行插值。我们只需要增加对 z 值进行插值的代码，并在内循环中加入 z 缓存逻辑即可。

1. 更新恒定 Shader 处理函数

在所有光栅化函数中，固定/恒定 Shader 处理函数的更新工作是最艰巨的，因为必须重新编写。作者将以 Gouraud Shader 处理函数为基础来完成这项工作：不进行 RGB 颜色插值，每个像素都使用相同的颜色，并将对某个颜色分量进行插值的代码改为对 z 值进行插值的代码。为此，需要完成两项工作。首先，必须确保新函数接受一个多边形数据结构作为参数，该多边形中包含每个顶点的 x、y、z 坐标。当前，光栅化函数没有以参数方式接受 z 坐标。下面是 T3DLIB7.CPP 中 Draw_RENDERLIST4DV2_Solid16()的部分代码：

```
void Draw_RENDERLIST4DV2_Solid16(RENDERLIST4DV2_PTR rend_list,
                 UCHAR *video_buffer,
                 int lpitch)
{
   if (!(rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_ACTIVE) ||
      (rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_CLIPPED ) ||
      (rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_BACKFACE) )
     continue; // 进入下一个多边形
// 需要首先检查是否带纹理，因为带纹理的多边形可能使用固定或恒定着色
// 以便决定调用哪个光栅化函数
if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
  {

  // 设置顶点坐标
```

```
  face.tvlist[0].x = (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
  face.tvlist[0].y = (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
  face.tvlist[0].u0 = (int)rend_list->poly_ptrs[poly]->tvlist[0].u0;
  face.tvlist[0].v0 = (int)rend_list->poly_ptrs[poly]->tvlist[0].v0;

  face.tvlist[1].x = (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
  face.tvlist[1].y = (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
  face.tvlist[1].u0 = (int)rend_list->poly_ptrs[poly]->tvlist[1].u0;
  face.tvlist[1].v0 = (int)rend_list->poly_ptrs[poly]->tvlist[1].v0;

  face.tvlist[2].x = (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
  face.tvlist[2].y = (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
  face.tvlist[2].u0 = (int)rend_list->poly_ptrs[poly]->tvlist[2].u0;
  face.tvlist[2].v0 = (int)rend_list->poly_ptrs[poly]->tvlist[2].v0;


  // 设置纹理
  face.texture = rend_list->poly_ptrs[poly]->texture;

  // 是否使用固定着色
  if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
   {
   // 以固定着色方式绘制带纹理的三角形
   Draw_Textured_Triangle16(&face, video_buffer, lpitch);
   } // end if
  else
   {
   // 使用恒定着色
   face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];
   Draw_Textured_TriangleFS16(&face, video_buffer, lpitch);
   } // end else

  } // end if
else
if ((rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_FLAT) ||
  (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_CONSTANT) )
  {
  // 使用基本的恒定着色光栅化函数绘制三角形
  Draw_Triangle_2D2_16(rend_list->poly_ptrs[poly]->tvlist[0].x,
            rend_list->poly_ptrs[poly]->tvlist[0].y,
         rend_list->poly_ptrs[poly]->tvlist[1].x,
            rend_list->poly_ptrs[poly]->tvlist[1].y,
       rend_list->poly_ptrs[poly]->tvlist[2].x,
            rend_list->poly_ptrs[poly]->tvlist[2].y,
         rend_list->poly_ptrs[poly]->lit_color[0],
              video_buffer, lpitch);

  } // end if
else
if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
  {
  // 设置顶点坐标
  face.tvlist[0].x = (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
  face.tvlist[0].y = (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
  face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];

  face.tvlist[1].x = (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
  face.tvlist[1].y = (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
  face.lit_color[1] = rend_list->poly_ptrs[poly]->lit_color[1];

  face.tvlist[2].x = (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
  face.tvlist[2].y = (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
  face.lit_color[2] = rend_list->poly_ptrs[poly]->lit_color[2];

  // 绘制使用 Gouraud 着色的三角形
  Draw_Gouraud_Triangle16(&face, video_buffer, lpitch);
  } // end if gouraud
```

```
 } // end for poly

} // end Draw_RENDERLIST4DV2_Solid16
```

该函数有三条主要的执行路径：光栅化使用固定/恒定着色处理的多边形、光栅化使用 Gouraud 着色处理的多边形以及光栅化使用纹理映射的多边形。对于每条执行路径，我们首先设置好参数，然后调用下述函数之一：

- **Draw_Gouraud_Triangle16()**——标准 Gouraud Shader；
- **Draw_Textured_Triangle16()**——支持固定着色的纹理映射函数；
- **Draw_Textured_TriangleFS16()**——支持恒定着色的纹理映射函数；
- **Draw_Triangle_2D2_16()**——对使用固定/恒定着色的多边形进行光栅化的老式函数。

我们将首先修改最后一个函数，但基本上所有这些函数都需要修改，包含 Draw_RENDERLIST4DV2_Solid16()。下面是对使用固定/恒定着色的多边形进行光栅化的函数的新原型：

```
void Draw_Triangle_2DZB_16(POLYF4DV2_PTR face, // 指向多边形面的指针
                UCHAR *_dest_buffer,  // 指向视频缓存的指针
                int mem_pitch,        // 每行占用多少个字节（320、640 等）
                UCHAR *zbuffer,       // 指向 z 缓存的指针
                int zpitch);          // Z 缓存每行占用的字节数
```

它不同于前一个版本。前一个版本接受顶点和颜色作为参数，而不是 POLYF4DV2_PTR。现在，我们需要统一调用方法，使所有光栅化函数的调用方法都相同。新增的其他参数包括指向 z 缓存的指针和 z 缓存的跨距（z 缓存中每行占用的字节数）。光栅化函数确保正确访问 z 缓存，而调用方必须确保 z 缓存足够大。该函数的调用方法如下：

```
// 使用固定着色
face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];

// 设置顶点坐标
face.tvlist[0].x = (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
face.tvlist[0].y = (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
face.tvlist[0].z = (int)rend_list->poly_ptrs[poly]->tvlist[0].z;

face.tvlist[1].x = (int)rend_list->poly_ptrs[poly]->tvlist[1].x;
face.tvlist[1].y = (int)rend_list->poly_ptrs[poly]->tvlist[1].y;
face.tvlist[1].z = (int)rend_list->poly_ptrs[poly]->tvlist[1].z;

face.tvlist[2].x = (int)rend_list->poly_ptrs[poly]->tvlist[2].x;
face.tvlist[2].y = (int)rend_list->poly_ptrs[poly]->tvlist[2].y;
face.tvlist[2].z = (int)rend_list->poly_ptrs[poly]->tvlist[2].z;

// 使用基本的恒定着色光栅化函数绘制三角形
Draw_Triangle_2DZB_16(&face, video_buffer, lpitch, zbuffer, zpitch);
```

在上述范例中，将渲染列表中一个多边形的顶点坐标复制到多边形面结构 face 中，多边形颜色被复制到 lit_color[0]中，光栅化使用恒定/固定着色的多边形时，只需要知道多边形的颜色。最后，使用视频缓存 video_buffer 及其跨距 lpitch 以及 z 缓存 zbuffer 及其跨距 zptich 来调用光栅化函数。读者对此应该很熟悉——编写光照/纹理映射模块后，就一直是这样调用其他光栅化函数的。

有关函数 Draw_Triangle_2DZB_16()及其调用方法就介绍到这里。该函数的代码太长，无法全部列出，这里只摘录绘制扫描线的代码，让读者知道是如何使用 z 缓存的：

```
// 将 screen_ptr 指向第一行
screen_ptr = dest_buffer + (ystart * mem_pitch);

// 将 z_ptr 指向第一行
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi<=yend; yi++)
  {
  // 计算扫描线的端点
  xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
```

```
xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

// 计算 z 的始值
zi = zl + FIXP16_ROUND_UP;

// 计算 dz
if ((dx = (xend - xstart))>0)
   {
   dz = (zr - zl)/dx;
   } // end if
else
   {
dz = (zr - zl);
} // end else

// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
    // 检查当前像素的 z 值是否小于 z 缓存中相应的值
    if (zi < z_ptr[xi])
     {
    // 绘制像素
     screen_ptr[xi] = color;

    // 更新 z 缓存
    z_ptr[xi] = zi;
    } // end if

  // 计算下一个像素的 z 值
   zi+=dz;
  } // end for xi

// 计算下一条扫描线起点和终点的 x 坐标和 z 坐标
xl+=dxdyl;
zl+=dzdyl;
xr+=dxdyr;
zr+=dzdyr;

// 将指针 screen_ptr 指向下一行
screen_ptr+=mem_pitch;

// 将指针 z_ptr 指向下一行
z_ptr+=zpitch;
} // end for y
```

上述代码绘制两个 y 值之间的多条扫描线，其中的 z 缓存代码为粗体，它们完全是按照前面介绍的方式做的。首先将当前 z 值同 z 缓存中相应的值进行比较：

```
if (zi < z_ptr[xi])
{
```

如果条件为真，则绘制像素：

```
// 绘制像素
screen_ptr[xi] = color;
```

并更新 z 缓存：

```
// 更新 z 缓存
z_ptr[xi] = zi;
} // end if
```

提示：在确定需要更新 z 缓存后，并没有立刻使用 zi 对其进行更新。这是一种优化技巧。读取某个值后立刻对其进行更新通常是一种糟糕的做法——最好在读写之间放置一些其他的代码。

这就是固定/恒定 Shader 的全部内容。除了增加了通过插值计算 z 值以及通过比较 z 值来更新 z 缓存的代码外，其他内容与前一版相同。

2. 更新 Gouraud Shader

在 Gouraud Shader 中加入 z 缓存支持很容易，因为它已经有插值计算 R、G、B 的代码，只需在后面加上插值计算 z 值的代码即可。新版本的原型如下，它只是名称不同，且新增了两个参数——z 缓存及其跨距：

```
void Draw_Gouraud_TriangleZB16(POLYF4DV2_PTR face,   // 指向多边形面的指针
        UCHAR *_dest_buffer,   // 指向视频缓存的指针
        int mem_pitch,     // 每行占用多少个字节（320、640 等）
        UCHAR *zbuffer,  // 指向 z 缓存的指针
        int zpitch);     // z 缓存中每行的字节数
```

该函数的调用方法与以前相同，只是需要提供指向 z 缓存的指针和 z 缓存的跨距（每行占据的字节数，和固定/恒定 Shader 中一样）。现在来看一下加入 z 缓存支持后，Gouraud Shader 中内循环的代码：

```
screen_ptr = dest_buffer + (ystart * mem_pitch);

// 将指针 z_ptr 指向起始行
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi<=yend; yi++)
  {
  // 计算扫描线端点的坐标
  xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
  xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

  // 计算 u、v、w 和 z 的起始值
  ui = ul + FIXP16_ROUND_UP;
  vi = vl + FIXP16_ROUND_UP;
  wi = wl + FIXP16_ROUND_UP;
  zi = zl + FIXP16_ROUND_UP;

  // 计算 u、v、w、z 的增量
  if ((dx = (xend - xstart))>0)
  {
  du = (ur - ul)/dx;
  dv = (vr - vl)/dx;
  dw = (wr - wl)/dx;
    dz = (zr - zl)/dx;
  } // end if
   else
  {
  du = (ur - ul);
  dv = (vr - vl);
  dw = (wr - wl);
  dz = (zr - zl);
  } // end else

//////////////////////////////////////////////////////////////////////////////

  // 左裁剪面裁剪测试
  if (xstart < min_clip_x)
  {
  // 计算 dx
  dx = min_clip_x - xstart;

  // 重新计算初值
  ui+=dx*du;
  vi+=dx*dv;
  wi+=dx*dw;
  zi+=dx*dz;

  // 重新设置变量
  xstart = min_clip_x;

  } // end if
```

```
  // 右裁剪面裁剪测试
  if (xend > max_clip_x)
    xend = max_clip_x;

////////////////////////////////////////////////////////////////////////////

  // 绘制扫描线
  for (xi=xstart; xi<=xend; xi++)
  {
    // 检查当前像素的 z 值是否小于 z 缓存中相应的值
    if (zi < z_ptr[xi])
     {
    // 绘制像素
      screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
              ((vi >> (FIXP16_SHIFT+2)) << 5) +
              (wi >> (FIXP16_SHIFT+3));

    // 更新 z 缓存
    z_ptr[xi] = zi;
    } // end if

// 计算下一个像素的 u、v、w、z
ui+=du;
vi+=dv;
wi+=dw;
zi+=dz;
} // end for xi
// 计算下一条扫描线起点和终点的 x、u、v、w、z
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;
wl+=dwdyl;
zl+=dzdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
wr+=dwdyr;
zr+=dzdyr;

// 将指针 screen_ptr 指向视频缓存的下一行
screen_ptr+=mem_pitch;

// 将指针 z_ptr 指向 z 缓存的下一行
z_ptr+=zpitch;

} // end for y
```

同样，对 z 值进行比较的代码用粗体显示。有趣的是，如果像素不用绘制，将节省一些开销。在前一个版本中，绘制像素的代码总是被执行；然而，在支持 z 缓存的版本中，仅当像素的 z 值小于 z 缓存中相应的值时，才绘制像素并更新 z 缓存。因此，为支持 z 缓存，我们需要做更多的工作，但当 z 缓存比较失败时，可节省时间。另外，为插值计算 z 值，每次外循环需要多计算两次加法运算：

```
zl+=dzdyl;
zr+=dzdyr;
```

同时，在每次内循环中需要多计算一次加法运算：

```
zi+=dz;
```

总之，加入 z 缓存支持对速度的影响并不大。事实上，如果运行演示程序，您将发现速度与没有使用 z 缓存时几乎一样快。但通过支持 z 缓存，对每个像素的渲染都是正确的。

3. 更新纹理映射函数

读者可能还记得，我们有两个纹理映射函数。一个用于处理使用固定着色的多边形，即根本不考虑光

照，只是将纹理图映射到多边形上。

另一个用于处理使用恒定着色且带纹理的多边形。这个纹理映射函数的名称带后缀 FS，它在执行光照计算时假定多边形的颜色为白色（即 RGB 值为(255，255，255)），并使用光照处理后的颜色对纹理图中每个像素进行调制。

（1）支持固定着色的纹理映射函数

我们需要在这两个纹理映射函数中加入 z 缓存支持，同样这也很简单。只需加入通过插值计算 z 值的代码，并在内循环中进行 z 缓存比较即可。下面是支持固定着色的纹理映射函数的新原型：

```
void Draw_Textured_TriangleZB16(POLYF4DV2_PTR face,  // 指向多边形面的指针
          UCHAR *_dest_buffer,  // 指向视频缓存的指针
          int mem_pitch,     // 每行多少字节（320、640 等）
          UCHAR *zbuffer,    // 指向 z 缓存的指针
          int zpitch);     // z 缓存每行多少字节
```

这个函数的调用方法与以前相同，只是需要设置 face 中的 z 坐标值，同时传递一个指向 z 缓存的指针以及该 z 缓存的跨距（每行占用多少个字节）。同样，这里只列出光栅化代码，其中 z 缓存比较代码以粗体显示：

```
// 将 screen_ptr 指向起始行
screen_ptr = dest_buffer + (ystart * mem_pitch);

// 将指针 z_ptr 指向起始行
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi<=yend; yi++)
   {
   // 计算扫描线端点的坐标
   xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
   xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

   // 计算 u、v 和 z 的起始值
   ui = ul + FIXP16_ROUND_UP;
   vi = vl + FIXP16_ROUND_UP;
   zi = zl + FIXP16_ROUND_UP;

   // 计算 u、v、z 的增量
   if ((dx = (xend - xstart))>0)
  {
  du = (ur - ul)/dx;
  dv = (vr - vl)/dx;
  dz = (zr - zl)/dx;
  } // end if
  else
  {
  du = (ur - ul);
  dv = (vr - vl);
  dz = (zr - zl);
  } // end else

////////////////////////////////////////////////////////////////////////////

  // 左裁剪面裁剪测试
  if (xstart < min_clip_x)
  {
  // 计算 dx
  dx = min_clip_x - xstart;

  // 重新计算初值
  ui+=dx*du;
  vi+=dx*dv;
  zi+=dx*dz;

  // 重新设置变量
  xstart = min_clip_x;
```

```
  } // end if

  // 右裁剪面裁剪测试
  if (xend > max_clip_x)
  xend = max_clip_x;

////////////////////////////////////////////////////////////////////////////

  // 绘制扫描线
  for (xi=xstart; xi<=xend; xi++)
   {
     // 检查当前像素的 z 值是否小于 z 缓存中相应的值
     if (zi < z_ptr[xi])
     {
     // 绘制像素
     screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
                  ((vi >> FIXP16_SHIFT) << texture_shift2)];

     // 更新 z 缓存
     z_ptr[xi] = zi;
     } // end if

    // 计算下一个像素的 u、v、z
     ui+=du;
     vi+=dv;
     zi+=dz;
     } // end for xi

// 计算下一条扫描线起点和终点的 x、u、v、z
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;
zl+=dzdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
zr+=dzdyr;

// 将指针 screen_ptr 指向视频缓存的下一行
screen_ptr+=mem_pitch;
  // 将指针 z_ptr 指向 z 缓存的下一行
  z_ptr+=zpitch;

} // end for y
```

加入 z 缓存支持是如此简单，让人有罪恶感。并非所有函数的编写工作都像实时彩色光照系统那么难。同样，除了在内循环和外循环中增加了一两次加法运算，用于插值计算 z 值外，其他内容完全相同。

（2）支持恒定着色的纹理映射函数

现在来看一下支持恒定着色的纹理映射函数。就代码方面而言，支持 z 缓存的 Gouraud Shader 函数是最复杂的，但就技术方面而言，支持恒定着色的纹理映射函数是最复杂的。我们需要一个能够接受 z 缓存及其跨距作为参数的新函数，其原型如下：

```
void Draw_Textured_TriangleFSZB16(POLYF4DV2_PTR face, // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少字节（320、640 等）
          UCHAR *zbuffer,    // 指向 z 缓存的指针
          int zpitch);     //z 缓存每行多少字节
```

这与前一版本相同，只是调用该函数之前，需要正确地设置多边形的 z 坐标，并在调用该函数时传递一个指向 z 缓存的指针以及该 z 缓存的跨距（每行占据的字节数）。下面是加入 z 缓存支持后，该函数中光栅化循环的代码，其中 z 缓存代码以粗体显示：

```
// 将 screen_ptr 指向起始行
```

```
screen_ptr = dest_buffer + (ystart * mem_pitch);

// 将指针 z_ptr 指向起始行
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi<=yend; yi++)
   {
   // 计算扫描线端点的坐标
   xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
   xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

   // 计算 u、v 和 z 的起始值
   ui = ul + FIXP16_ROUND_UP;
   vi = vl + FIXP16_ROUND_UP;
   zi = zl + FIXP16_ROUND_UP;

   // 计算 u、v、z 的增量
   if ((dx = (xend - xstart))>0)
  {
  du = (ur - ul)/dx;
  dv = (vr - vl)/dx;
  dz = (zr - zl)/dx;
  } // end if
   else
  {
  du = (ur - ul);
  dv = (vr - vl);
    dz = (zr - zl);
  } // end else

   // 绘制扫描线
   for (xi=xstart; xi<=xend; xi++)
  {
    //  检查当前像素的 z 值是否小于 z 缓存中相应的值
     if (zi < z_ptr[xi])
      {
     //  绘制像素
     //  首先获取纹素
    textel = textmap[(ui >> FIXP16_SHIFT) +
             ((vi >> FIXP16_SHIFT) << texture_shift2)];

     // 提取 R、B、G 分量
     r_textel = ((textel >> 11)    );
     g_textel = ((textel >> 5) & 0x3f);
     b_textel =  (textel    & 0x1f);
     // 用光照处理后的颜色调制纹素
     r_textel*=r_base;
     g_textel*=g_base;
     b_textel*=b_base;
     // 最后绘制像素
     // 我们将 R、G、B 分别扩大了 32、64 和 32 倍
     // 因此首先需要将它们分别除以 32、64 和 32
     // 但由于要将它们进行移位，以便组成一个 5.6.5 字
     // 因此在移位时一并完成
     screen_ptr[xi] = ((b_textel >> 5) +
             ((g_textel >> 6) << 5) +
             ((r_textel >> 5) << 11));

     // 更新 z 缓存
     z_ptr[xi] = zi;
     } // end if

  //  计算下一个像素的 u、v、z
  ui+=du;
    vi+=dv;
    zi+=dz;
  } // end for xi
```

```
// 计算下一条扫描线起点和终点的 x、u、v、z
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;
zl+=dzdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
zr+=dzdyr;

// 将指针 screen_ptr 指向视频缓存的下一行
screen_ptr+=mem_pitch;

// 将指针 z_ptr 指向 z 缓存的下一行
z_ptr+=zpitch;

} // end for y
```

内循环比较复杂，在 z 缓存比较失败时不绘制像素是件好事；在这种情况下，可避免执行很多代码，将跳过下述计算：

```
// 绘制像素
// 首先获取纹素
textel = textmap[(ui >> FIXP16_SHIFT) +
         ((vi >> FIXP16_SHIFT) << texture_shift2)];

//提取 R、B、G 分量
r_textel = ((textel >> 11)    );
g_textel = ((textel >> 5) & 0x3f);
b_textel =  (textel    & 0x1f);

// 用光照处理后的颜色调制纹素
r_textel*=r_base;
g_textel*=g_base;
b_textel*=b_base;
screen_ptr[xi] = ((b_textel >> 5) +
         ((g_textel >> 6) << 5) +
         ((r_textel >> 5) << 11));

// 更新 z 缓存
z_ptr[xi] = zi;
```

有趣的是，随着光栅化函数越来越复杂，我们以为它们的速度会降低，但结果却是更快，因为将其用于真正的游戏世界时得大于失。在这里，通过 z 缓存比较可避免绘制某些像素，这抵消了插值计算对性能的降低；其结果是，像素得到了正确的渲染，而没有付出任何代价。

4. 更新渲染列表函数

最后一个需要加入 z 缓存功能的函数是渲染列表函数。这个函数接受渲染列表作为参数，根据渲染列表中多边形的类型调用相应的光栅化函数。下面是新版本的渲染列表绘制函数：

```
void Draw_RENDERLIST4DV2_SolidZB16(RENDERLIST4DV2_PTR rend_list,
                   UCHAR *video_buffer,
                    int lpitch,
                   UCHAR *zbuffer,
                   int zpitch)
{
// 16 位版本
// 这个函数绘制渲染列表中所有的多边形
// 它根据多边形的光照模型调用相应的光栅化函数


POLYF4DV2 face; // 用于存储要渲染的多边形
```

```
for (int poly=0; poly < rend_list->num_polys; poly++)
  {
  // 当且仅当多边形没有被剔除和裁剪掉、处于活动状态并可见时才渲染它
  // 但在线框引擎中，背面的概念无关紧要
  if (!(rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_ACTIVE) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_CLIPPED ) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_BACKFACE) )
    continue; // 进入下一个多边形

  //需要首先检查是否带纹理，因为带纹理的多边形可能使用固定或恒定着色
  // 以便决定调用哪个光栅化函数
  if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
    {

    // 设置顶点坐标
    face.tvlist[0].x  = (float)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y  = (float)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z  = (float)rend_list->poly_ptrs[poly]->tvlist[0].z;
    face.tvlist[0].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[0].u0;
    face.tvlist[0].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[0].v0;

    face.tvlist[1].x  = (float)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y  = (float)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z  = (float)rend_list->poly_ptrs[poly]->tvlist[1].z;
    face.tvlist[1].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[1].u0;
    face.tvlist[1].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[1].v0;

    face.tvlist[2].x  = (float)rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y  = (float)rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z  = (float)rend_list->poly_ptrs[poly]->tvlist[2].z;
    face.tvlist[2].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[2].u0;
    face.tvlist[2].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[2].v0;

    // 设置纹理
    face.texture = rend_list->poly_ptrs[poly]->texture;

    // 是否使用固定着色
    if (rend_list->poly_ptrs[poly]->attr &
      POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
     {
     // 以固定着色方式绘制带纹理的三角形
     Draw_Textured_TriangleZB16(&face, video_buffer,
                    lpitch,zbuffer,zpitch);
     } // end if
    else
     {
     // 使用恒定着色
     face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];
     Draw_Textured_TriangleFSZB16(&face, video_buffer,
                    lpitch,zbuffer,zpitch);
     } // end else

    } // end if
  else
  if ((rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_FLAT) ||
    (rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_CONSTANT) )
    {
    // 使用固定着色
    face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];

    // 设置顶点坐标
    face.tvlist[0].x = (float)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y = (float)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z = (float)rend_list->poly_ptrs[poly]->tvlist[0].z;

    face.tvlist[1].x = (float)rend_list->poly_ptrs[poly]->tvlist[1].x;
```

```
  face.tvlist[1].y = (float)rend_list->poly_ptrs[poly]->tvlist[1].y;
  face.tvlist[1].z = (float)rend_list->poly_ptrs[poly]->tvlist[1].z;

  face.tvlist[2].x = (float)rend_list->poly_ptrs[poly]->tvlist[2].x;
  face.tvlist[2].y = (float)rend_list->poly_ptrs[poly]->tvlist[2].y;
  face.tvlist[2].z = (float)rend_list->poly_ptrs[poly]->tvlist[2].z;

  // 使用基本的恒定着色光栅化函数绘制三角形
  Draw_Triangle_2DZB_16(&face, video_buffer, lpitch,zbuffer,zpitch);

  } // end if
else
if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
  {
  // 设置顶点坐标
  face.tvlist[0].x  = (float)rend_list->poly_ptrs[poly]->tvlist[0].x;
  face.tvlist[0].y  = (float)rend_list->poly_ptrs[poly]->tvlist[0].y;
  face.tvlist[0].z  = (float)rend_list->poly_ptrs[poly]->tvlist[0].z;
  face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];

  face.tvlist[1].x  = (float)rend_list->poly_ptrs[poly]->tvlist[1].x;
  face.tvlist[1].y  = (float)rend_list->poly_ptrs[poly]->tvlist[1].y;
  face.tvlist[1].z  = (float)rend_list->poly_ptrs[poly]->tvlist[1].z;
  face.lit_color[1] = rend_list->poly_ptrs[poly]->lit_color[1];

  face.tvlist[2].x  = (float)rend_list->poly_ptrs[poly]->tvlist[2].x;
  face.tvlist[2].y  = (float)rend_list->poly_ptrs[poly]->tvlist[2].y;
  face.tvlist[2].z  = (float)rend_list->poly_ptrs[poly]->tvlist[2].z;
  face.lit_color[2] = rend_list->poly_ptrs[poly]->lit_color[2];

  // 绘制使用 Gouraud 着色的三角形
  Draw_Gouraud_TriangleZB16(&face, video_buffer, lpitch, zbuffer, zpitch);
  } // end if gouraud
 } // end for poly

} // end Draw_RENDERLIST4DV2_SolidZB16
```

请读者花些时间研读它，其中每条控制路径都清晰而简单。这个函数检查当前多边形的类型，然后调用相应的光栅化函数。当然，如果使用 C++，只需对函数进行重载或使用虚函数，根本不需要使用 case 语句。然而，C++对于固定设计来说很不错，但我们的设计是零乱的，因此使用 C++并没有多大帮助。我们只是编写代码，试验算法，并查看结果而已。以后，读者编写代码来实现这些功能时，无疑需要以更有效、更清晰的方式进行。

函数 Draw_RENDERLIST4DV2_SolidZB16()的调用方法与其前身 Draw_RENDERLIST4DV2_Solid16()相同，只是新增了两个参数：指向 z 缓存的指针以及该 z 缓存的跨距（每行占用多少个字节）。

注意：读者可能会问，为何不将指向 z 缓存结构的指针 ZBUFFERV1_PTR 传递给光栅化函数呢？不这样做旨在确保开放性，如果以后决定对 z 缓存结构 ZBUFFERV1 进行升级，将不会影响到渲染函数。

11.4　可能的 z 缓存优化

在光栅化期间对 z 缓存值的计算已经很简单，没有必要再使用汇编语言（SIMD 指令等）。因此，我们的着眼点应该是更高级的优化。有几种可能的优化，下面来介绍它们。

11.4.1　使用更少的内存

第一种让 z 缓存的速度更高些的方法是使其更小些，即使用 16 位而不是 32 位的 z 缓存。这看起来是个不错的主意，但存取速度可能比 32 位缓存更慢。原因在于奔腾处理器是 32 位的设备，更擅强于每次处

理 32 位。虽然每行占据的内存越少，高速缓存的连贯性（coherence）越高，但由于对齐方式（alignment）和其他微妙的因素，16 位存取可能降低性能。另外，为确定像素的前后顺序，用 16 位来表示 z 值已接近最低限度了，这是另一个需要考虑的问题。作者发现，由于 z 缓存比较位于光栅化函数的内循环中，使用 16 位的 z 缓存对性能的提高帮助不大。

11.4.2 降低清空 z 缓存的频率

清空 z 缓存需要一些时间。基本上，它相当于将一个 32 位值写入内存所需时间乘以 z 缓存的大小。例如，假设使用的是奔腾 III，由于高速缓存和其他作用，它写一次内存需要 4 个时钟周期；再假设时钟频率为 500MHz，则一个时钟周期为 2×10^{-9} 秒（2 纳秒）。如果 z 缓存大小为 800×600，则清空 z 缓存需要的时间为：

```
总时间 =
clocks_per_write * width * height * time_per_clock
= 4*800*600*(2x10-9 s) = 3.84 毫秒
```

这看似不长，但考虑到 60 帧/秒时每帧只持续 16 毫秒，这已经很长了。这意味着有 24%（3.84/16）的时间被用于清空 z 缓存。这是有害的，必须避免。为此，有几种办法：可以使用矩形记录当前帧不涉及的 z 缓存区域，然后不对这些区域进行清空。这听起来不错，但实际上并不管用。另一种方法是根本不清空 z 缓存。然而，这只能使用 1/z 缓存才能实现。由于使用 1/z 缓存时，只要 z>=1，1/z 的值就不可能大于 1.0，我们可以在每帧中都给 1/z 加上一个偏移量，并在更新 1/z 缓存时，使用包含该偏移量的 1/z。这样，在每一帧中，任何像素的 1/z 都比以前各帧中任何像素的 1/z 大。另外，还需要修改 1/z 缓存的比较方式，即仅当像素的 1/z 大于 1/z 缓存中相应的值时，才绘制该像素。图 11.10 说明了这种方法。

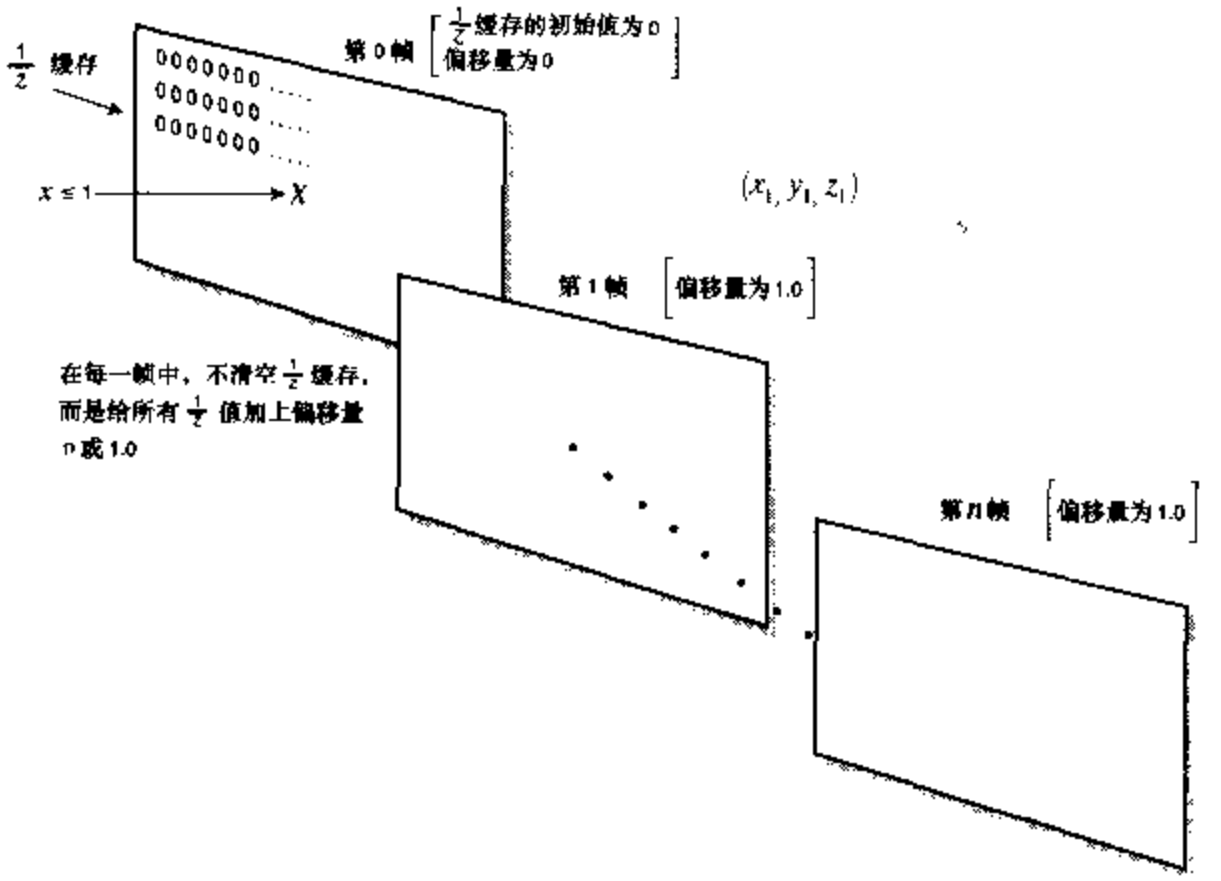


图 11.10 避免清空 1/z 缓存

作为一个例子，这里使用这种技术来绘制两帧。每个像素的 x 和 y 坐标都相同，因此只需要考虑它们的 z 坐标。在每一帧中，都有一系列 x 和 y 坐标相同但 z 坐标不同的像素，以检查这种 1/z 缓存技术是否管用。如果给 1/z 加上偏移量，下一帧将被正确渲染。下面进行验证。

渲染第 0 帧时，1/z 缓存的初始值为 0。各个像素的情况如下：

像素编号	z	1/z	偏移量	最终 1/z（包括偏移量）	像素编号	z	1/z	偏移量	最终 1/z（包括偏移量）
0	10	.100	0	.100	2	5	.200	0	.200
1	12	.083	0	.083					

下面分析第 0 帧的渲染过程。

像素 0 的最终 1/z 为 0.1，大于 0.0，因此被绘制到屏幕上，同时将 1/z 缓存中对应的值修改为 0.1。

像素 1 的最终 1/z 为 0.083，小于 0.1，因此不绘制它，同时 1/z 缓存保持不变。

像素 2 的最终 1/z 为 0.2，大于 0.1，因此绘制它，并将 1/z 缓存中相应的值修改为 0.2。

从上述分析可知，最后被绘制到屏幕上的是离视点最近的像素（像素 2，其 z 值为 5.0）。

渲染第 1 帧时，1/z 缓存中的值为 0.2。各个像素的情况如下：

像素编号	z	1/z	偏移量	最终 1/z（包括偏移量）	像素编号	z	1/z	偏移量	最终 1/z（包括偏移量）
0	5	.200	1.0	1.200	2	3	.333	1.0	1.333
1	2	.5	1.0	1.500					

下面分析第 1 帧的渲染过程。

像素 0 的最终 1/z 为 1.2，大于 0.2，因此绘制它，同时将 1/z 缓存中对应的值修改为 1.2。

像素 1 的最终 1/z 为 1.5，大于 1.2，因此绘制它，同时 1/z 缓存中对应的值修改为 1.5。

像素 2 的最终 1/z 为 1.33，小于 1.5，因此不绘制它，同时 1/z 缓存保持不变。

从上述分析可知，最后被绘制到屏幕上的是离视点最近的像素（像素 1，其 z 值为 2.0）。

为避免写入延迟，1/z 缓存很不错。惟一的问题是，在某个时刻，1/z 缓存会溢出。这发生在第 n 帧之后，n 为 1/z 缓存在除掉用于存储小数部分后的位后，能存储的最大整数。

11.4.3　混合 z 缓存

下一个优化方法是使用混合 z 缓存。也就是说，这对场景的某一部分使用 z 缓存，对其他部分使用 z 排序。例如，可以对 z 值大于视距 30%的所有多边形使用 z 排序，因为在大多数时候，这样做不会影响效果；对于 z 值小于视距 30%的多边形，使用 z 缓存。实际上，这样做的效果非常好。我们首先对较远的多边形进行 z 排序，使用画家算法按从后到前的顺序渲染它们；然后，对于要求顺序绝对正确的较近的多边形，使用 z 缓存。图 11.11 说明了这种方法。

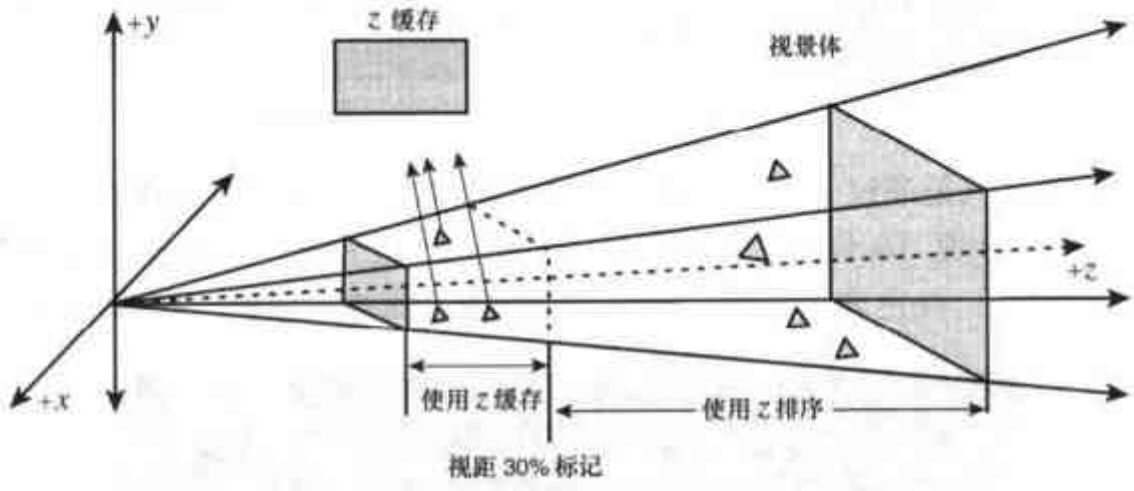


图 11.11　混合使用 z 缓存和 z 排序

11.5　z 缓存存在的问题

最后要说一下 z 缓存的问题。显然，第一个问题是它根本不正确。我们在屏幕空间中使用线性插值来计算并不呈线性变化的值，因此将得到扭曲的结果，但这不是主要问题。z 缓存的主要问题在于，经过透视变换后，z 值并非线性变化的，当视景体非常大时，这将降低精度。换句话说，假设在游戏空间中，远裁剪面离视点的距离为 1 000 000 个单位（对于太空游戏，这完全是合理的）；为准确地表示 1 000 000 个单位，需要 20 位，即使 z 缓存是 32 位的，也只余下 12 位可用于表示 z 值的小数部分。为表示这么大的 z 值范围，降低了 z 缓存的精度。这是 1/z 缓存更好的另一个原因。

11.6 软件和 z 缓存演示程序

首先来说一下软件模块。本章所需的函数和常量都包含在下述库文件中：

- **T3DLIB9.CPP**——C/C++源代码；
- **T3DLIB9.H**——T3DLIB9.CPP 的头文件。

当然，要编译本章的演示程序，除上述两个文件外，还需要 T3DLIB1.CPP|H 到 T3DLIB8.CPP|H。

11.6.1 演示程序 I：z 缓存可视化

首先来看一个技术演示程序。图 11.12 是演示程序 DEMOII11_1.CPP|EXE 的屏幕截图。使用箭头键可以缓慢地移动环境，使用 Z 键可以选择 z 排序或 z 缓存。请选择其中之一，并移动环境使其与玩家物体（player object）相互贯穿；然后在 z 排序和 z 缓存之间来回切换，以了解使用 z 缓存的效果有多么完美。这很好，但如果能够将 z 缓存可视化就更好了。

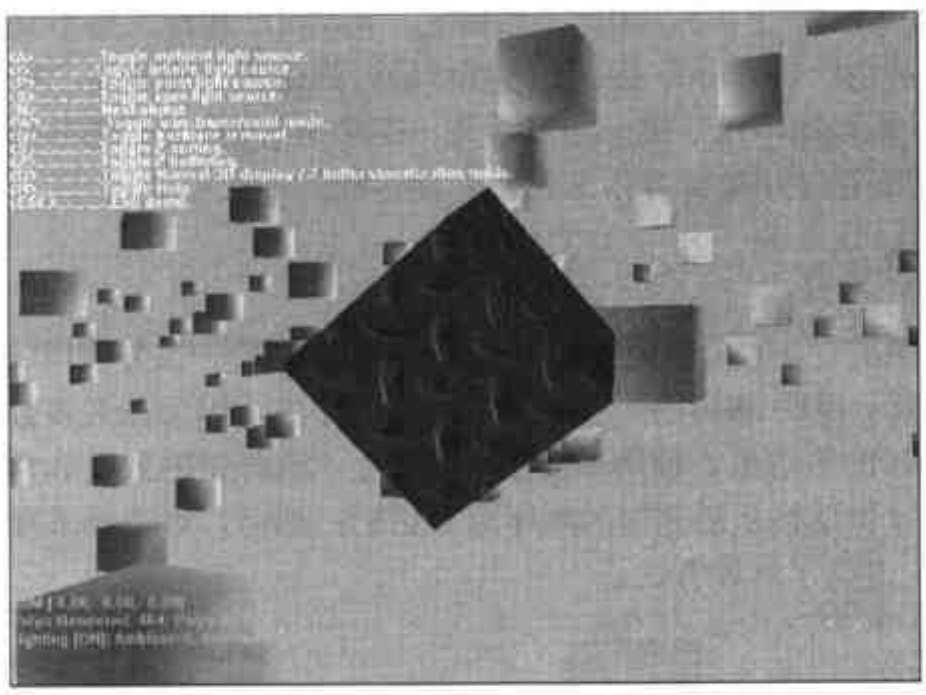


图 11.12 第一个 z 缓存演示程序的屏幕截图

这不难。基本上，作者只是通过一个简单的转换过滤器将 z 缓存复制到帧缓存中，该过滤器将 32 位的 z 值转换为颜色。由于 RGB 是 16 位的，其中高位表示的是红色分量，低位表示的是蓝色分量，这使得 z 缓存相当于一个彩色深度图，红色表示离视点较远，蓝色表示离视点较近，如图 11.13 所示。

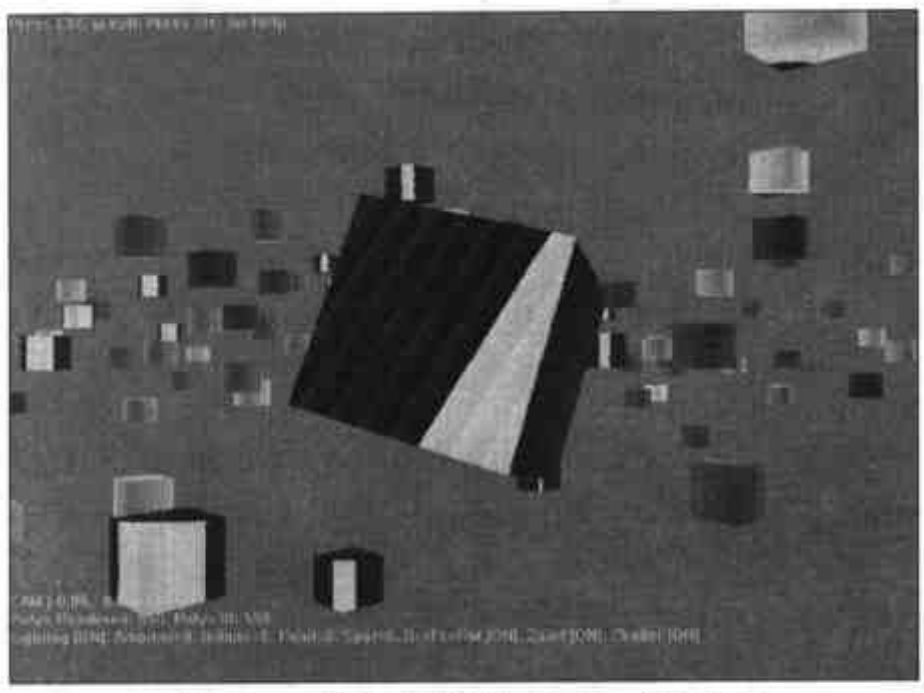


图 11.13 z 缓存可视化模式下的 z 缓存演示

这很不错，它让作者想起了电影 *Predator*。下面是一些重要的控制方法：

- 箭头键——移动；
- Z——开/关 *z* 缓存；
- S——开/关 *z* 排序；
- N——选择物体；
- H——开/关帮助菜单；
- D——在常规渲染和 *z* 缓存过滤器之间切换；
- Esc——退出演示程序。

要编译该演示程序，需要主.CPP 文件 DEMOII11_1.CPP、库模块 T3DLIB1.CPP|H 到 T3DLIB9.CPP|H，还需要链接 DirectX 库和 WINMM.LIB。

警告：在 *z* 缓存可视化模式下，将发现颜色是循环的。这是因为 z 值线性地从 0 变化到无穷大，而颜色为 5.6.5 格式的 RGB 值，将 z 值映射到颜色时，当 z 从 0 增加到 65536 时，最后 5 位（表示蓝色分量）将循环 2048 次。然后，总体而言，从蓝色变为红色表示深度增大。

11.6.2　演示程序 II：Wave Raider

为编写这个演示程序，作者以前一章的地形演示程序为基础，对物理模型进行了改进，并进行了水面模拟。*z* 缓存非常适合用于水面，因为“水面多边形”被绘制在某个水平面上，而摩托艇多边形可穿透水面，看起来部分在水面上，部分在水下。图 11.14 是演示程序 DEMOII11_2.CPP|EXE 的屏幕截图；相对于使用的代码量而言，这很不错。

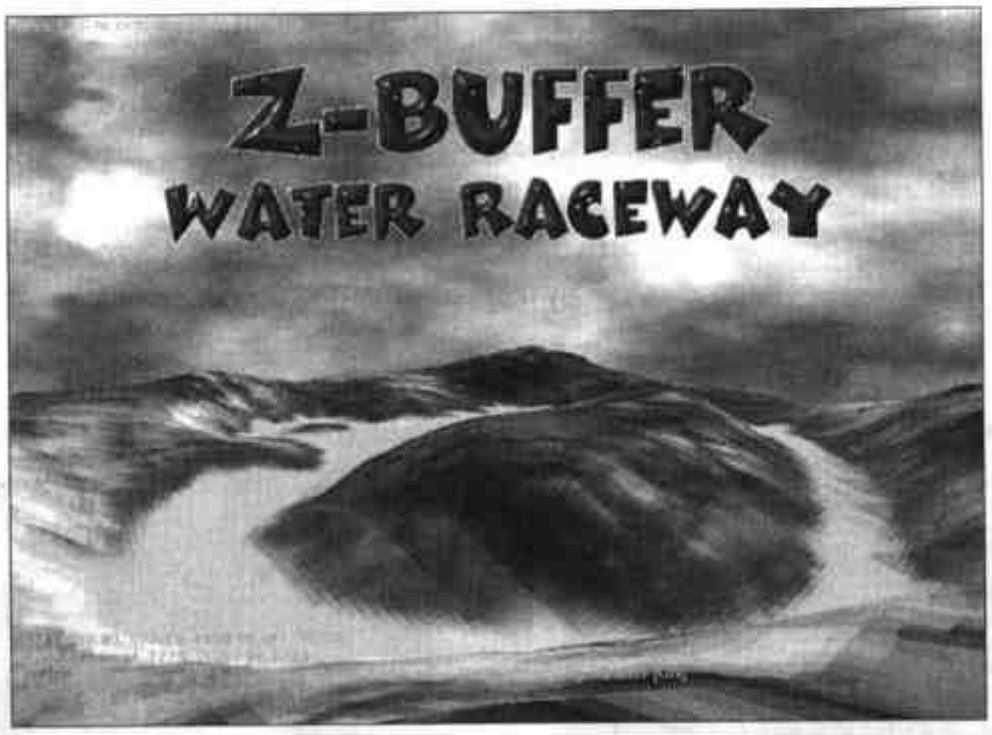


图 11.14　Wave Raider（非常适合使用 *z* 缓存来实现穿透水面的效果）

下面是该演示程序的控制方式：

- 回车——发动发动机；
- 右箭头——右转；
- 左箭头——左转；
- 上箭头——踩油门。

以前使用的大部分帮助菜单和引擎控制方式仍然管用：

- Z——开/关 *z* 缓存；
- S——开/关 *z* 排序；
- D——在常规渲染方式和 *z* 缓存过滤器之间切换；
- H——开/关帮助菜单；
- Esc——退出演示程序。

接下来介绍该演示程序的 3 个编程要点。

1. 水面赛道

为创建赛道，作者经历了一场小小的恶梦。能够创建高程图的地形生成程序很多，但作者实在太忙，不可能花好几个小时去弄清楚如何使用这些工具来创建赛道。最后，作者找到了两个很有用的程序：

- Terragen——http://www.planetside.co.uk/terragen/；
- Worldmachine——http://students.washington.edu/sschmitt/world/。

这两个程序都有读者可以使用的共享软件版本。作者最后使用 Worldmachine 为这个游戏演示程序创建了高程图。图 11.15 是 Worldmachine 的屏幕截图。

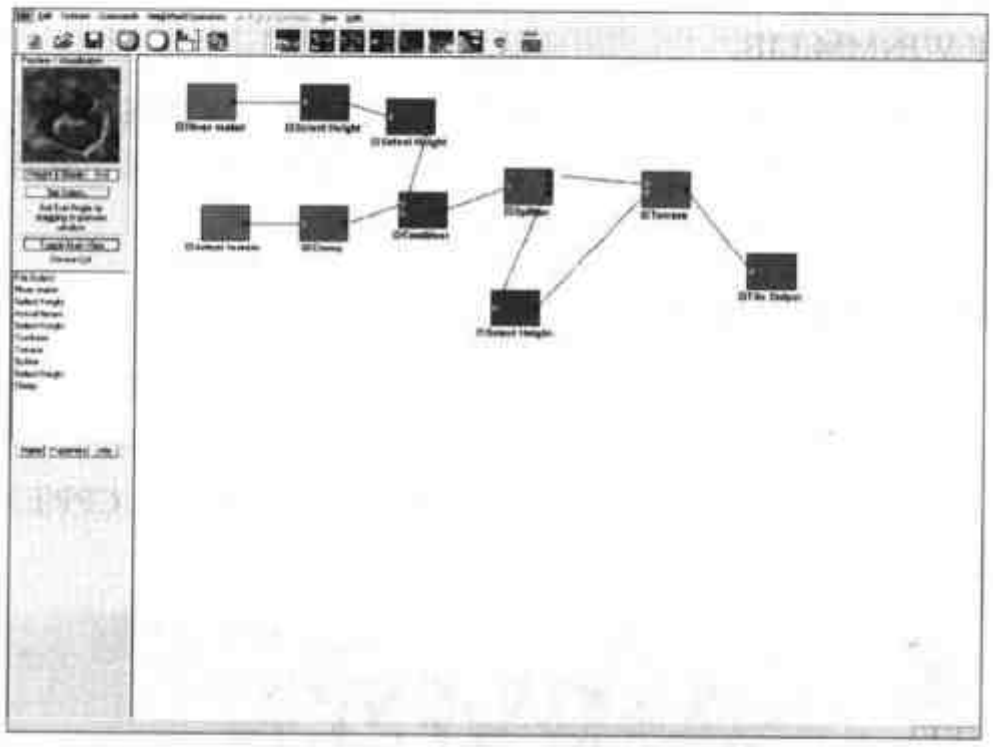


图 11.15　使用 Worldmachine 创建水上赛道

为创建地形，需要使用该程序创建一个高程图和一个纹理图。图 11.16 是灰度格式的高程图，图 11.17 是 Worldmachine 生成的纹理图。高程图是 40×40 的；而纹理图是 256×256 的，但图 11.16 显示的是分辨率更高（512×512）的版本，以便读者能够看清楚。

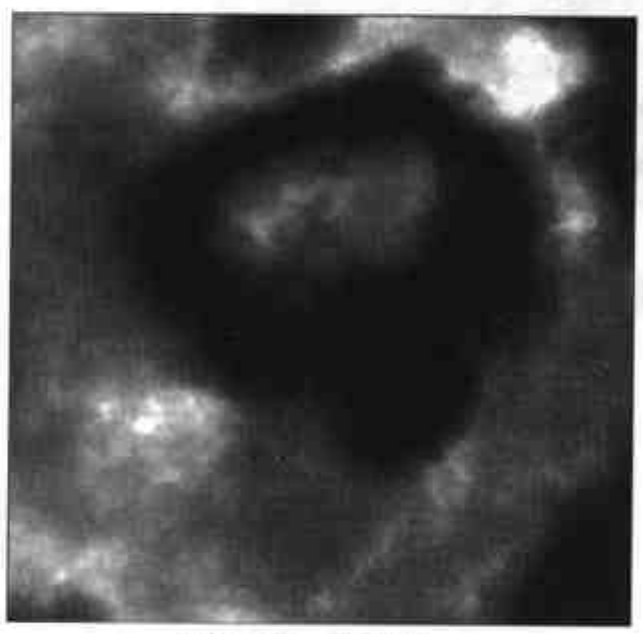


图 11.16　高程图



图 11.17　水上赛道的纹理图

为生成地形做好准备后，调用前一章中的地形生成函数，如下所示：

```
// 生成地形
Generate_Terrain_OBJECT4DV2(&obj_terrain,      // 指向物体的指针
  TERRAIN_WIDTH,        // 地形在世界坐标系中的宽度（沿 x 轴）
  TERRAIN_HEIGHT,       // 地形在世界坐标系中的长度（沿 z 轴）
  TERRAIN_SCALE,        // 地形的最大可能高度
  "water_track_height_04.bmp", // 256 色高程图的文件名
```

```
"water_track_color_03.bmp",  // 纹理图的文件名
RGB16Bit(255,255,255), // 没有纹理时地形的颜色
&terrain_pos,       // 初始位置
NULL,           // 初始旋转角度
POLY4DV2_ATTR_RGB16 |
POLY4DV2_ATTR_SHADE_MODE_FLAT | /*POLY4DV2_ATTR_SHADE_MODE_GOURAUD */
POLY4DV2_ATTR_SHADE_MODE_TEXTURE );
```

正如读者看到的，生成的地形使用恒定着色。

2. 水波模拟

水波模拟其实很简单，只需使用正弦或其他相关的波函数对地形进行调制（modulate）即可，秘诀在于只调制有水的部分。如果这样考虑，问题并不复杂：在每帧的开始，将用局部坐标表示的初始地形复制到变换后的地形中，以便能够对其进行修改。然后对整个地形运行波函数（稍后将介绍），但只对位于虚拟水平面之下的顶点进行调制。图 11.18 说明了这一点。

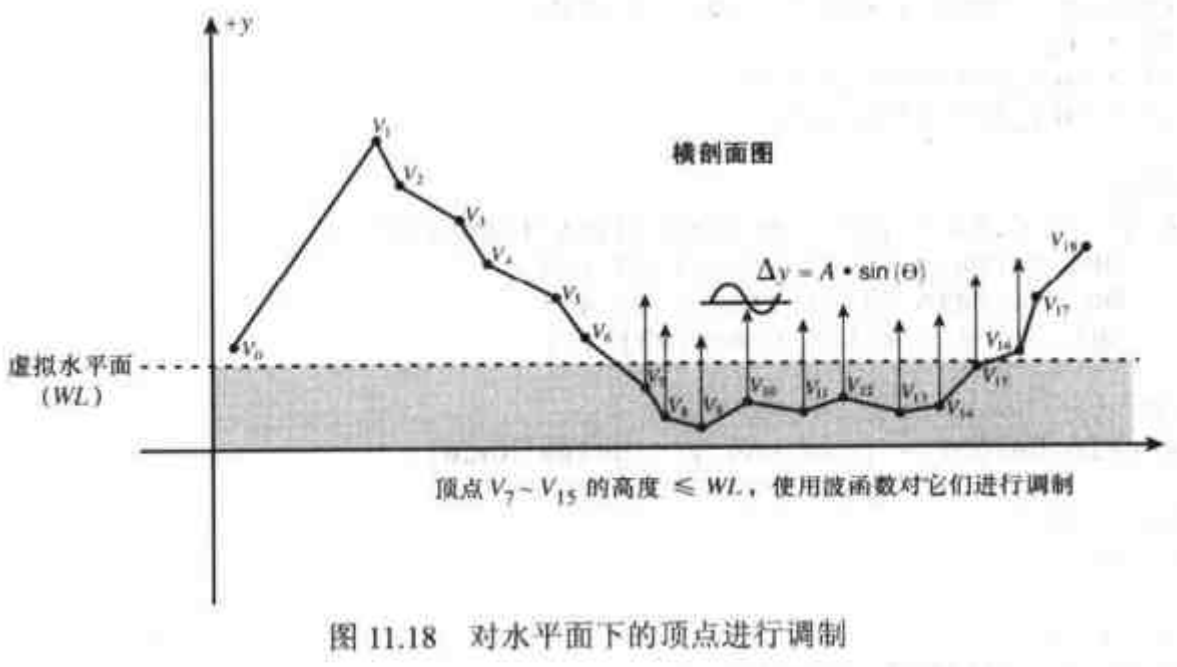


图 11.18　对水平面下的顶点进行调制

下面是对地形中高度小于水平面的顶点进行调制的代码：

```
// 水波生成算法 ////////////////////////////////////////////////////

// 使用波函数对地形中每个高度低于水平线的顶点进行调制
for (int v = 0; v < obj_terrain.num_vertices; v++)
  {
  // 只对高度低于水平线的顶点进行调制
  if (obj_terrain.vlist_trans[v].y < WATER_LEVEL)
    obj_terrain.vlist_trans[v].y+=
     WAVE_HEIGHT*sin(wave_count + (float)v/(2*(float)obj_terrain.ivar2+0));
  } // end for v

// 提高改变波形位置
wave_count+=WAVE_RATE;
```

读者可能认为，涉及弹簧常数和波函数等，物理模型一定很复杂。并非如此。只需要一个正弦波和一个计数器。并非编写水波模型不重要，只是有时候可以使用简单的代码来实现特殊效果。

3. 摩托艇的物理模型

摩托艇的物理模型与前一章的沙地汽车物理模型相同。基本上，只需修改常量，使摩托艇更有弹性，能够更深入地穿透水面，以演示 z 缓存的用途。从很大程度上说，几乎对物理模型没有做任何修改。玩家启动摩托艇后，地形跟踪算法根据地形情况，将摩托艇往上推或让它下落。有趣的地方在于，通过使用波函数对地形进行调制，使摩托艇看起来随水波上下运动。下面是实现物理模型的代码，摩托艇被放置在相机的前面：

```
// 运动部分 ////////////////////////////////////////////////////

// 地形跟踪算法确定当前所在的单元格，然后计算其 4 个顶点处的平均高度
```

```
// 并根据沙地汽车的当前和地形的平均高度，将摩托艇往上推

// 一些用于地形跟踪的变量
//ivar1 = columns;
//ivar2 = rows;
//fvar1 = col_vstep;
//fvar2 = row_vstep;

int cell_x = (cam.pos.x + TERRAIN_WIDTH/2) / obj_terrain.fvar1;
int cell_y = (cam.pos.z + TERRAIN_HEIGHT/2) / obj_terrain.fvar1;
static float terrain_height, delta;

// 检查是否在地形上
if ( (cell_x >=0) && (cell_x < obj_terrain.ivar1) &&
   (cell_y >=0) && (cell_y < obj_terrain.ivar2) )
  {
  // 计算当前单元格顶点的索引
  int v0 = cell_x + cell_y*obj_terrain.ivar2;
  int v1 = v0 + 1;
  int v2 = v1 + obj_terrain.ivar2;
  int v3 = v0 + obj_terrain.ivar2;

  // 计算平均高度
  terrain_height = 0.25 * (obj_terrain.vlist_trans[v0].y +
              obj_terrain.vlist_trans[v1].y +
              obj_terrain.vlist_trans[v2].y +
              obj_terrain.vlist_trans[v3].y);

  // 计算高度差
  delta = terrain_height - (cam.pos.y - gclearance);

  // 检测是否穿入水面
  if (delta > 0)
   {
   // 给相机一个作用力
   vel_y+=(delta * (VELOCITY_SCALER));

   // 检测是否穿入水面，如果是，立刻向上移动
   // 以免穿透几何体
   cam.pos.y+=(delta*CAM_HEIGHT_SCALER);

   // 根据前进速度和地形坡度调整仰角
   cam.dir.x -= (delta*PITCH_CHANGE_RATE);

   } // end if

  } // end if

// 减速
if (cam_speed > (CAM_DECEL) ) cam_speed-=CAM_DECEL;
else
if (cam_speed < (-CAM_DECEL) ) cam_speed+=CAM_DECEL;
else
  cam_speed = 0;

// 将相机的方向限制在一定范围内
if (cam.dir.x > (neutral_pitch+PITCH_RETURN_RATE))
   cam.dir.x -= (PITCH_RETURN_RATE);
else
if (cam.dir.x < (neutral_pitch-PITCH_RETURN_RATE))
   cam.dir.x += (PITCH_RETURN_RATE);
  else
   cam.dir.x = neutral_pitch;

// 应用重力
vel_y+=gravity;
```

```
// 检查是否在海平面下，如果是，将其置于海平面上
if (cam.pos.y < sea_level)
  {
  vel_y = 0;
  cam.pos.y = sea_level;
  } // end if

// 移动相机
cam.pos.x += cam_speed*Fast_Sin(cam.dir.y);
cam.pos.z += cam_speed*Fast_Cos(cam.dir.y);
cam.pos.y += vel_y;

// 将点光源放置在摩托艇前面
lights2[POINT_LIGHT_INDEX].pos.x = cam.pos.x + 150*Fast_Sin(cam.dir.y);
lights2[POINT_LIGHT_INDEX].pos.y = cam.pos.y + 50;
lights2[POINT_LIGHT_INDEX].pos.z = cam.pos.z + 150*Fast_Cos(cam.dir.y);

// 将摩托艇放在相机前面的水下
obj_player.world_pos.x = cam.pos.x + 120*Fast_Sin(cam.dir.y);
obj_player.world_pos.y = cam.pos.y - 75 + 7.5*sin(wave_count);
obj_player.world_pos.z = cam.pos.z + 120*Fast_Cos(cam.dir.y);
```

编写真正的游戏时，必须让相机始终对准玩家，而不是将相机作为参照系，将玩家置于相机前面。另外，这个物理模型没有摩擦力，因此摩托艇转向的速度非常快。要在物理模型中考虑摩擦力很容易，但作者需要将一些内容留给其他演示程序！

4. 音响效果

作者对这个游戏中的音响非常满意。作者将摩托艇、水浪、海洋、水流飞溅以及自己（作为解说员）的声音混合在一起，给玩家以玩游戏的体验。音响效果由三部分组成，下面简要地介绍它们。在游戏中，音响效果非常重要，可能成为游戏有趣还是乏味的分水岭。

（1）解说员

该游戏有 3 个主要部分：解说、下水发令枪响和驾驶摩托艇。启动游戏后，首先是一段解说，然后是摩托艇下水，并有一段解说。

为制作画外音，作者自己进行解说，使用 Sound Forge 以 11kHz、8 位单声道方式进行录制；同时加入一些喧闹声，以增加真实感；最后加入回音。最终效果很不错，朋友们都不敢相信这是我的声音。

（2）水声

水声取自 Sound Ideas General 6000 声音库，这个声音库由 40 张 CD 组成，花了作者大约 2500 美金，不过每个子都花得值。作者使用了较高和较低的两种水波声，以不同的音量并根据摩托艇的速度修改频率。令人惊讶的是，采用 8 位和 11kHz 的采样方式时，水浪声基本上不受影响。当然，水浪声是循环播放的。

（3）摩托艇声音

这是最复杂的部分。使用摩托艇声音旨在营造摩托艇呼啸而过的效果。在现实世界中，很多东西并不发出声音——我们只是想象它发出声音。例如，在电影中，声音被夸张到荒谬的程度。艺术家拍完电影后进行后期制作，旨在让观众看到画面的同时听到声音，即使本来没有声音。

摩托艇有下列几种状态：

- 状态 1——发动机熄火；
- 状态 2——发动机启动（一次）；
- 状态 3——空档（循环）；
- 状态 4——加速（一次）；
- 状态 5——发动机以恒定的速度运转（循环）。

需要发动机启动的声音，作者将 25-HP 外置马达的声音和摩托艇启动的声音混合起来，因为摩托艇启动的声音太小。在空档、加速时使用摩托艇的声音。复杂的地方在于，如何正确地将它们进行混合。为此，作者根据摩托艇的状态、速度以及其他变量来更换声音。例如，当玩家按回车键启动摩托艇时，播放除加速外的其他所有声音。玩家只能听到发动机发动的声音，因为其声音非常大，空档时的声明被覆盖了。然

而，此时没有使用状态 5 时的声音，因为摩托艇还是静止的。

当玩家踩油门时，播放加速声音一次，玩家将感觉到摩托艇开始前进。同时，将增加摩托艇的速度。根据摩托艇的速度来调整发动机转动声的频率和音量，速度越快，颠簸越严重，声音越大。玩家松开油门后，将播放空档时的声音。

当然，模拟这些声音的方式很多，除了调整音量外，还有其他地方需要优化；但当前的效果确实不错。

警告：光栅化函数仍存在除零 bug，当退化的三角形被传递给光栅化函数时，演示程序将崩溃。这种情况发生在穿过地形最高处时——那里有青草，位于玩家的左边。

要编译该演示程序，除了主.CPP 文件 DEMOII11_2.CPP 外，还需要库模块 T3DLIB1.CPP|H 到 T3DLIB9.CPP|H，同时需要链接 DirectX 库和 WINMM.LIB。

11.7 总　　结

另一个看似复杂的主题又被打得跷地求饶了。学习图形学的技巧是，花些时间在草稿纸上演算范例，理解它，然后所有一切都能解决。现在，我们有了一个相当不错的 3D 引擎，它支持 16 位模式、光照计算、z 缓存和模型加载；速度也相当快，根本不需要优化。以后将继续在引擎中添加功能：完善光照计算、支持层次型物体（如室内关卡）等。从很大程度上说，读者已经掌握了所需 3D 图形学知识的 90%，应感到自豪。

第四部分

高级 3D 渲染

第 12 章　高级纹理映射技术

第 13 章　空间划分和可见性算法

第 14 章　阴影和光照映射

第 12 章　高级纹理映射技术

本章将重点讨论光栅化技术。虽然我们的光栅化内核已经相当高级，但它还不支持 alpha 混合、透视修正纹理映射、纹理滤波、mipmapping、使用 Gouraud 着色处理的纹理映射等。本章将处理这些问题，我们将重新编写所有的光栅化函数。下面是本章将介绍的重要主题：

- 透明度和 alpha 混合；
- 使用 Gouraud 着色处理的纹理映射；
- 透视修正纹理映射；
- 1/z 缓存；
- 双线性纹理滤波；
- mipmapping 和三线性纹理滤波；
- 新的物体加载函数；
- 多次渲染。

12.1　纹理映射——第二波

本章编写的光栅化函数如此之多，作者都感觉要爆炸了。仅光栅化函数，代码就有大约 50 000 行，这是因为对于加入到引擎中的每项功能，都编写了一个独立的光栅化函数——作者不想使用通用的光栅化函数，以免降低性能。

编写这些光栅化函数后，提供了一些演示程序。作者认识到，不适合在本章深入探讨光照和阴影——这完全是另一个主题。另外，作者确实想介绍光映射、环境映射和阴影的使用，这需要可用于创建关卡的东西，因此将这些主题推迟到下一章再介绍。在该章中，将介绍空间划分，这样将有关卡编辑工具，供编写光映射演示程序时使用。

本章将介绍很多新技术，如 1/z 缓存、alpha 混合、透视修正纹理映射、mipmapping、双线性和三线性纹理映射等。当然，需要重新编写所有的光栅化函数，本章新增的光栅化函数有 40 多个，编码和调试的工作量很大。作者竭力确保所有光栅化函数的速度都很快，读者将对其速度感到惊讶，而这都是在没有使用汇编语言和进行过多优化的情况下实现的。优化将留到第 16 章去完成。

和以往一样，要编译本章的程序，需要下述新增的库模块和头文件：

- T3DLIB10.CPP——C/C++源文件；
- T3DLIB10.H——头文件。

要编译本章的程序，除上述文件外，还需要包含以前的库模块：T3DLIB1-9.CPP|H；另外还需要链接 DirectX .LIB 文件，并正确的设置编译器（生成 Win32 .EXE 文件）。在源代码的开头，都指出了编译程序所需的文件。

在本章中，对于每个主题都将进行简要的介绍，列出重要的实现代码，创建一个演示程序，然后进入下一个主题。有些技术（如三线性滤波）没有实现（或者实现后发现其速度太慢）。

另外，为支持新增的特性，对其他一些函数进行了更新，如地形生成函数、Caligari trueSpace .COB 文件加载函数。和往常一样，这里首先快速地浏览一下头文件，让读者对本章的内容有大致了解。如果读者不能理解这些代码，也不用担心，这里旨在让读者对本章的内容有大致了解。

头文件

下面介绍头文件 T3DLIB10.H。虽然本章后面将详细地介绍这些函数和数据结构，但作者喜欢从介绍全章涉及的全部函数和数据结构开始。首先介绍常量。本章新定义的常量不多，只是定义了一些与定点数运算相关的常量以及一组渲染选项，后面编写一个包罗万象的渲染函数时，需要使用这些渲染选项。

```
// DEFINES ////////////////////////////////////////////////////////////

// 用于 alpha 混合的常量
#define NUM_ALPHA_LEVELS          8   // alph 等级数

// 用于 1/z 缓存的常量
#define FIXP28_SHIFT             28  // 用于 1/z 缓存
#define FIXP22_SHIFT             22  // 用于透视修正纹理映射中使用的 u/z 和 v/z

// 用于支持 mipmapping 的新属性
#define POLY4DV2_ATTR_MIPMAP          0x0400 // 多边形是否有 mip 纹理链
#define OBJECT4DV2_ATTR_MIPMAP        0x0008 // 物体是否有 mip 纹理链

//////////////////////////////////////////////////////////////////////

// 用于控制渲染函数状态的属性
// 每类控制标记都存储在一个 4 位字节中，这有助于以后扩展

// 不使用 z 缓存，按多边形在渲染列表中的顺序渲染它们
#define RENDER_ATTR_NOBUFFER                  0x00000001

// 使用 z 缓存
#define RENDER_ATTR_ZBUFFER                   0x00000002

// 使用 1/z 缓存
#define RENDER_ATTR_INVZBUFFER                0x00000004

// 使用 mipmapping
#define RENDER_ATTR_MIPMAP                    0x00000010

// 启用 alpha 混合
#define RENDER_ATTR_ALPHA                     0x00000020

// 启用双线性滤波，但只适用于固定着色/仿射纹理映射函数
#define RENDER_ATTR_BILERP                    0x00000040

// 使用仿射纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE   0x00000100

// 使用完美透视修正纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT  0x00000200

// 使用线性分段透视纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR   0x00000400

// 根据距离混合使用仿射纹理映射和线性分段纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1  0x00000800

// 还未实现
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID2  0x00001000
```

接下来是一个新增的数据结构，作者称之为渲染场境（rendering context）。使用所需的渲染选项对它进行初始化后，便可以将其作为参数来调用一个可渲染任何东西的函数，而不必像以前那样调用大量不同的渲染函数。下面是这个数据结构的定义：

```
// 这是一种用于表示渲染场境的数据类型
// 我们可以根据需要设置这种数据结构，然后将其作为参数传递给渲染函数
// 从而避免将越来越多的参数传递给渲染函数
typedef struct RENDERCONTEXTV1_TYP
{
int     attr;          // 渲染属性
RENDERLIST4DV2_PTR rend_list; // 指向要渲染的渲染列表的指针
UCHAR *video_buffer;      // 指向视频缓存的指针
int     lpitch;        // 视频缓存的跨距，单位为字节

UCHAR  *zbuffer;          // 指向 z 缓存或 1/z 缓存的指针
int     zpitch;         // z 缓存或 1/z 缓存的跨距
int     alpha_override;     // 用于重新设置所有多边形的 alpha 值

int     mip_dist;          // 最大 mip 距离
int     texture_dist,    // 使用透视/仿射混合模式时，启用仿射纹理映射的距离

// 供以后扩展时使用
int     ival1, ivalu2;
float   fval1, fval2;
void    *vptr;

} RENDERCONTEXTV1, *RENDERCONTEXTV1_PTR;
```

其中的大部分字段读者都不会感到陌生，只有几个字段是新出现的，它们用于支持 mipmapping 和各种经过优化的纹理映射技术，将在讨论相关主题时进行介绍。

接下来介绍全局变量，这里只有一个：帮助执行 alpha 混合的查找表：

```
// 这个查找表中包含各种 RGB 值与某些标量的乘积
extern USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536];
```

最后，介绍函数原型，让读者对本章将编写多少个函数以及将完成的工作有大致的了解：

```
// 函数原型 ////////////////////////////////////////////////////////
// 使用 z 缓存的函数版本
void Draw_Textured_TriangleZB2_16(POLYF4DV2_PTR face,  // 指向多边形面的指针
              UCHAR *_dest_buffer, // 指向视频缓存的指针
              int mem_pitch,      // 每行多少个字节（320、640 等）
              UCHAR *zbuffer,      // 指向 z 缓存的指针
              int zpitch);        // z 缓存中每行多少个字节

void Draw_Textured_Bilerp_TriangleZB_16(POLYF4DV2_PTR face,// 指向面的指针
         UCHAR *_dest_buffer, // 指向视频缓存的指针
         int mem_pitch,     // 每行多少个字节（320、640 等）
         UCHAR *zbuffer,     // 指向 z 缓存的指针
         int zpitch);       // z 缓存中每行多少个字节

void Draw_Textured_TriangleFSZB2_16(POLYF4DV2_PTR face, // 指向面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少个字节（320、640 等）
          UCHAR *zbuffer,     // 指向 z 缓存的指针
          int zpitch);       // z 缓存中每行多少个字节

void Draw_Textured_TriangleGSZB_16(POLYF4DV2_PTR face,   // 指向面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少个字节（320、640 等）
          UCHAR *_zbuffer,    // 指向 z 缓存的指针
          int zpitch);       // z 缓存中每行多少个字节

void Draw_Triangle_2DZB2_16(POLYF4DV2_PTR face, // 指向面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少个字节（320、640 等）
          UCHAR *zbuffer,    // 指向 z 缓存的指针
          int zpitch);       // z 缓存中每行多少个字节

void Draw_Gouraud_TriangleZB2_16(POLYF4DV2_PTR face,   // 指向面的指针
```

```
            UCHAR *_dest_buffer,// 指向视频缓存的指针
            int mem_pitch,   // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,   // 指向 z 缓存的指针
            int zpitch);    // z 缓存中每行多少个字节

void Draw_RENDERLIST4DV2_SolidZB2_16(RENDERLIST4DV2_PTR rend_list,
        UCHAR *video_buffer, //指向视频缓存的指针
        int lpitch, // 每行多少个字节（320、640 等）
        UCHAR *zbuffer, // 指向 z 缓存的指针
        int zpitch); // z 缓存中每行多少个字节


// 1/z 缓存版本

void Draw_Textured_TriangleINVZB_16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,     // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Textured_Bilerp_TriangleINVZB_16(POLYF4DV2_PTR face,//指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,     // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Textured_TriangleFSINVZB_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,     // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Textured_TriangleGSINVZB_16(POLYF4DV2_PTR face,   // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,   // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Triangle_2DINVZB_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,     // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Gouraud_TriangleINVZB_16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Textured_Perspective_Triangle_INVZB_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Textured_Perspective_Triangle_FSINVZB_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer,// 指向视频缓存的指针
            int mem_pitch,    // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,   // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节

void Draw_Textured_PerspectiveLP_Triangle_FSINVZB_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节
```

```
void Draw_Textured_PerspectiveLP_Triangle_INVZB_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch);       // z 缓存中每行多少个字节

void Draw_RENDERLIST4DV2_SolidINVZB_16(RENDERLIST4DV2_PTR rend_list,
            UCHAR *video_buffer,
          int lpitch,
            UCHAR *zbuffer,
            int zpitch);

// 支持 z 缓存和 alpha 混合
void Draw_Textured_TriangleZB_Alpha16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,      // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_TriangleFSZB_Alpha16(POLYF4DV2_PTR face,// 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,     // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_TriangleGSZB_Alpha16(POLYF4DV2_PTR face,// 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Triangle_2DZB_Alpha16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            CHAR *zbuffer,     // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Gouraud_TriangleZB_Alpha16(POLYF4DV2_PTR face,   // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_RENDERLIST4DV2_SolidZB_Alpha16(RENDERLIST4DV2_PTR rend_list,
            UCHAR *video_buffer,
            int lpitch,
              UCHAR *zbuffer,
              int zpitch,
              int alpha_override);

// 支持 1/z 缓存和 alpha 混合
void Draw_Textured_TriangleINVZB_Alpha16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_TriangleFSINVZB_Alpha16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
```

```
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_TriangleGSINVZB_Alpha16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Triangle_2DINVZB_Alpha16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Gouraud_TriangleINVZB_Alpha16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_Perspective_Triangle_INVZB_Alpha16(POLYF4DV2_PTR face,
            UCHAR *_dest_buffer,// 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_PerspectiveLP_Triangle_INVZB_Alpha16(POLYF4DV2_PTR face,
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_Perspective_Triangle_FSINVZB_Alpha16(POLYF4DV2_PTR face,
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_Textured_PerspectiveLP_Triangle_FSINVZB_Alpha16(POLYF4DV2_PTR face,
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少个字节
            int alpha);

void Draw_RENDERLIST4DV2_SolidINVZB_Alpha16(RENDERLIST4DV2_PTR rend_list,
            UCHAR *video_buffer,
          int lpitch,
            UCHAR *zbuffer,
            int zpitch,
            int alpha_override);

// 不使用缓存的函数
void Draw_Textured_Triangle2_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer,  // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Textured_Bilerp_Triangle_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
```

```
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Textured_TriangleFS2_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Triangle_2D3_16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer,// 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Gouraud_Triangle2_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Textured_TriangleGS_16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Textured_Perspective_Triangle_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Textured_PerspectiveLP_Triangle_16(POLYF4DV2_PTR face,
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Textured_Perspective_Triangle_FS_16(POLYF4DV2_PTR face,
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_Textured_PerspectiveLP_Triangle_FS_16(POLYF4DV2_PTR face,
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);    // 每行多少个字节（320、640 等）

void Draw_RENDERLIST4DV2_Solid2_16(RENDERLIST4DV2_PTR rend_list,
             UCHAR *video_buffer,
          int lpitch);

// 不使用缓存但支持 alpha 混合的函数
int RGB_Alpha_Table_Builder(int num_alpha_levels,   // alpha 等级数
     USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536]); // 查找表

void Draw_Triangle_2D_Alpha16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            int alpha);
void Draw_Textured_Triangle_Alpha16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            int alpha);

void Draw_Textured_TriangleFS_Alpha16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            int alpha);

void Draw_Textured_TriangleGS_Alpha16(POLYF4DV2_PTR face,  // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            int alpha);

void Draw_Gouraud_Triangle_Alpha16(POLYF4DV2_PTR face,   // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            int alpha);

void Draw_RENDERLIST4DV2_Solid_Alpha16(RENDERLIST4DV2_PTR rend_list,
```

```
            UCHAR *video_buffer, //指向视频缓存的指针
           int lpitch, //每行多少个字节(320、640 等)
            int alpha_override); //alpha 值

// 新的渲染函数
void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16(RENDERCONTEXTV1_PTR rc);

// 新的模型加载函数
int Load_OBJECT4DV2_COB2(OBJECT4DV2_PTR obj, // 指向物体的指针
             char *filename,    // Caligari COB 文件的名称
             VECTOR4D_PTR scale, // 初始缩放因子
             VECTOR4D_PTR pos,   // 初始位置
             VECTOR4D_PTR rot,   // 初始旋转角度
             int vertex_flags,   // 顶点顺序反转和变换标记
             int mipmap=0);      // mipmapping 启用标记
                      //1 表示生成 mip 纹理，0 表示不生成

int Generate_Terrain2_OBJECT4DV2(OBJECT4DV2_PTR obj,   // 指向物体的指针
    float twidth,      // 地形宽度
    float theight,      // 地形长度
    float vscale,      // 最大可能高度
    char *height_map_file, // 256 色高程图的文件名
    char *texture_map_file,// 纹理图的文件名
    int rgbcolor,      // 没有纹理时地形的颜色
    VECTOR4D_PTR pos,   // 初始位置
    VECTOR4D_PTR rot,   // 初始旋转角度
    int poly_attr,     // Shader
    float sea_level=-1,  // 海平面高度
    int alpha=-1);    // 海平面下的多边形的 alpha 值

// mipmapping 函数
int Generate_Mipmaps(BITMAP_IMAGE_PTR source,
           BITMAP_IMAGE_PTR *mipmaps, float gamma = 1.01);
int Delete_Mipmaps(BITMAP_IMAGE_PTR *mipmaps, int leave_level_0);
```

12.2　新的光栅化函数

首先要重新编写的是核心渲染函数。我们已经有一组渲染函数，但它们存在一些问题。其中一个主要问题是，在渲染函数中调用三角形光栅化函数时，总是将顶点的各种值截尾成整数，而不是将其保留为浮点数，让光栅化函数以更精确的方式将其转换为定点数。这样做降低了精度。下面是渲染函数将顶点的值传递给三角形光栅化函数之前执行的典型操作：

```
// 设置顶点信息
face.tvlist[0].x = (int)rend_list->poly_ptrs[poly]->tvlist[0].x;
face.tvlist[0].y = (int)rend_list->poly_ptrs[poly]->tvlist[0].y;
```

之所以需要进行截尾，主要原因是用于纹理映射的定点光栅化函数需要使用整型坐标。然而，这样做降低了精度。现在，这个问题得到了解决——所有三角形光栅化函数都接受浮点数作为参数，然后进行四舍五入，将其转换为定点数。

12.2.1　最终决定使用定点数

作者决定使用定点数。在大多数情况下，浮点数的乘除运算比定点数快，但问题是，光栅化从本质上说是一种“位”运算，处理的是 RGB 值、纹理坐标和内存。在这方面，浮点数非常糟糕，在光栅化函数的外循环和内循环对扫描线进行光栅化时，需要在浮点数和整数之间进行转换，其开销是无法承担的。在最糟糕的情况下，浮点数到整数的转换需要 70 个时钟周期，这是完全不能接受的。另外，使用 32 位整数时，可以进行各种使用浮点数无法进行的优化。

因此，除光栅化外，在其他情况下都将使用浮点数。这是最好的选择：在变换和光照计算中使用浮点

数，速度更快，精度更高；但在光栅化期间，使用浮点数的开销是无法承担的，因此改用定点数。作者做过试验：编写浮点数、定点数和混合版本的光栅化函数，最终发现 32 位定点数版本的速度要快得多。

然而，为提高小数运算的精度，不能仅使用以前的 16.16 定点数格式，还应补充其他格式，如 4.28 和 10.22，这将在讨论 1/z 缓存和透视修正纹理映射时介绍，它们需要使用新的定点数格式。

总之，从现在开始，我们将在所有光栅化函数中使用定点数；另外，还将在光栅化函数中更严格地遵守左上填充规则。现在，它们的精度超过像素级，纹理映射和实体渲染非常准确。在多边形相邻区域，不会由于错误的填充规则而出现裂缝或重复绘制像素。下面来看一下加入新特性后的光栅化函数。

12.2.2 不使用 z 缓存的新光栅化函数

首先来看一下不使用 z 缓存的光栅化函数。当然，还新增了支持透视修正、alpha 混合和 mipmapping 的光栅化函数，但这里只介绍在原有光栅化函数的基础上修改而成的光栅化函数。下面列出这些函数的原型，并做简要地说明：

```
void Draw_Triangle_2D3_16(POLYF4DV2_PTR face,  // 指向面的指针
        UCHAR *_dest_buffer,// 指向视频缓存的指针
        int mem_pitch);    // 每行多少个字节（320、640 等）
```

该函数将使用固定或恒定着色的多边形绘制到屏幕上。它不使用 z 缓存，因此在绘制多边形时不考虑渲染顺序，调用方必须预先对多边形进行排序。在原型方面，除了名称不同外，该函数与 2.0 版完全相同。然而更快、更清晰，精度超过像素级，遵循了左上填充规则。另外，前一个版本将三角形划分成两个：一个平顶三角形和一个平底三角形；但现在它与其他光栅化函数一样，将三角形作为常规三角形进行处理。

```
void Draw_Gouraud_Triangle2_16(POLYF4DV2_PTR face, // 指向面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch);   // 每行多少个字节（320、640 等）
```

该函数在前一版本的基础上进行了改进，现在精度超过像素级，更严格地遵守了左上填充规则。

```
void Draw_Textured_Triangle2_16(POLYF4DV2_PTR face, // 指向面的指针
         UCHAR *_dest_buffer,  // 指向视频缓存的指针
         int mem_pitch);     // 每行多少个字节（320、640 等）
```

该函数是在以前的仿射纹理映射函数的基础上改进而成的，现在精度超过像素级，更严格地遵守了左上填充规则。这改善了纹理映射效果，使其看起来更为平滑。

注意：老实说，仿射纹理映射的效果很好，速度很快，作者都不想使用透视修正纹理映射了。

```
void Draw_Textured_TriangleFS2_16(POLYF4DV2_PTR face, // 指向面的指针
           UCHAR *_dest_buffer, // 指向视频缓存的指针
           int mem_pitch);    // 每行多少个字节（320、640 等）
```

该函数是在支持恒定着色的仿射纹理映射函数的基础上改进而成的，现在精度超过像素级，更严格地遵守了左上填充规则。这改善了纹理映射效果，使其看起来更为平滑。

```
void Draw_RENDERLIST4DV2_Solid2_16(RENDERLIST4DV2_PTR rend_list, //绘制渲染列表
            UCHAR *video_buffer, //指向视频缓存的指针
            int lpitch); //每行多少个字节(320、640 等)
```

该函数是调用上述函数（不使用 z 缓存的光栅化函数）的入口。除了不再执行截尾操作外，代码与前一个版本相同，如下所示。

```
void Draw_RENDERLIST4DV2_Solid2_16(RENDERLIST4DV2_PTR rend_list,
                  UCHAR *video_buffer,
                   int lpitch)
{
// 16 位版本
// 该函数绘制渲染列表中所有的多边形
// 它根据多边形的光照模型调用相应的光栅化函数

POLYF4DV2 face; // 用于存储当前要渲染的多边形
```

```
for (int poly=0; poly < rend_list->num_polys; poly++)
  {
  // 仅当多边形没有被裁剪和剔除掉、处于活动状态且可见时才渲染它
  if (!(rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_ACTIVE) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_CLIPPED ) ||
     (rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_BACKFACE) )
    continue; // 进入下一个多边形

  // 首先检测是否带纹理，因为带纹理的多边形可能使用固定或恒定着色
  // 需要据此调用不同的光栅化函数
  if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
    {

    // 设置顶点信息
    face.tvlist[0].x = (float)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y = (float)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z  = (float)rend_list->poly_ptrs[poly]->tvlist[0].z;
    face.tvlist[0].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[0].u0;
    face.tvlist[0].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[0].v0;

    face.tvlist[1].x = (float)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y = (float)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z  = (float)rend_list->poly_ptrs[poly]->tvlist[1].z;
    face.tvlist[1].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[1].u0;
    face.tvlist[1].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[1].v0;

    face.tvlist[2].x = (float)rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y = (float)rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z  = (float)rend_list->poly_ptrs[poly]->tvlist[2].z;
    face.tvlist[2].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[2].u0;
    face.tvlist[2].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[2].v0;


    // 指定纹理
    face.texture = rend_list->poly_ptrs[poly]->texture;

    // 三角形使用固定着色?
    if (rend_list->poly_ptrs[poly]->attr &
      POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
     {
     // 使用固定着色来绘制带纹理的三角形
     Draw_Textured_Triangle2_16(&face, video_buffer, lpitch);
     //Draw_Textured_Perspective_Triangle_16(&face, video_buffer, lpitch);
     } // end if
    else
    if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_FLAT)
     {
     // 使用恒定着色
     face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];
     Draw_Textured_TriangleFS2_16(&face, video_buffer, lpitch);
     //Draw_Textured_Perspective_Triangle_FS_16(&face, video_buffer, lpitch);
     } // end else
    else
     {
     // 必定使用 Gouraud 着色
     face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];
     face.lit_color[1] = rend_list->poly_ptrs[poly]->lit_color[1];
     face.lit_color[2] = rend_list->poly_ptrs[poly]->lit_color[2];
     Draw_Textured_TriangleGS_16(&face, video_buffer, lpitch);
     } // end else

    } // end if
else
if ((rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_FLAT) ||
 (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_CONSTANT) )
  {
  // 使用固定着色处理
```

```
    face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];

    // 设置顶点信息
    face.tvlist[0].x = (float)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y = (float)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z  = (float)rend_list->poly_ptrs[poly]->tvlist[0].z;

    face.tvlist[1].x = (float)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y = (float)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z  = (float)rend_list->poly_ptrs[poly]->tvlist[1].z;

    face.tvlist[2].x = (float)rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y = (float)rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z  = (float)rend_list->poly_ptrs[poly]->tvlist[2].z;

    // 调用基本光栅化函数来绘制三角形
    Draw_Triangle_2D3_16(&face, video_buffer, lpitch);

    } // end if
  else
  if (rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
    {
    // 设置顶点信息
    face.tvlist[0].x  = (float)rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y  = (float)rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z  = (float)rend_list->poly_ptrs[poly]->tvlist[0].z;
    face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];

    face.tvlist[1].x  = (float)rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y  = (float)rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z  = (float)rend_list->poly_ptrs[poly]->tvlist[1].z;
    face.lit_color[1] = rend_list->poly_ptrs[poly]->lit_color[1];

    face.tvlist[2].x  = (float)rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y  = (float)rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z  = (float)rend_list->poly_ptrs[poly]->tvlist[2].z;
    face.lit_color[2] = rend_list->poly_ptrs[poly]->lit_color[2];

    // 使用 Gouraud 着色来绘制三角形
    Draw_Gouraud_Triangle2_16(&face, video_buffer, lpitch);

    } // end if gouraud

  } // end for poly

} // end Draw_RENDERLIST4DV2_Solid2_16
```

现在使用的是 float：按原样设置顶点的值，不会降低精度。下面来看一下使用 z 缓存的光栅化函数，请暂时将 alpha 混合等新技术抛到脑后。

12.2.3 支持 z 缓存的新光栅化函数

下面列出支持 z 缓存的光栅化函数，其中不包括使用 alpha 混合的函数，因为还没有介绍这种技术。这些函数只是用于替换以前的版本。

```
void Draw_Triangle_2DZB2_16(POLYF4DV2_PTR face, // 指向面的指针
        UCHAR *_dest_buffer, // 指向视频缓存的指针
        int mem_pitch,     // 每行多少个字节（320、640 等）
        UCHAR *zbuffer,    // 指向 z 缓存的指针
        int zpitch);      // z 缓存中每行多少个字节
```

在使用 z 缓存的光栅化函数中，这个函数是最基本的，它使用 z 缓存将三角形绘制到屏幕上，并相应地更新 z 缓存。该函数的精度超过了像素级，并更严格地遵守了左上填充规则。除名称不同外，参数与前一版相同。

```
void Draw_Textured_TriangleZB2_16(POLYF4DV2_PTR face,  // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
```

```
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节
```

该函数使用 z 缓存绘制使用固定着色和仿射纹理映射的三角形，其效果比前一个版本更好，因为它的精度超过像素级，且遵守了左上填充规则。同样，调用方法与前一版本相同。

```
void Draw_Textured_TriangleFSZB2_16(POLYF4DV2_PTR face, // 指向面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少个字节
```

该函数使用 z 缓存绘制使用恒定着色和仿射纹理映射的三角形，其效果比前一个版本更好，因为它的精度超过像素级，且遵守了左上填充规则。同样，调用方法与前一版本相同。

```
void Draw_Gouraud_TriangleZB2_16(POLYF4DV2_PTR face,   // 指向面的指针
            UCHAR *_dest_buffer,// 指向视频缓存的指针
            int mem_pitch,    // 每行多少个字节（320、640 等）
            UCHAR *zbuffer,   // 指向 z 缓存的指针
            int zpitch);     // z 缓存中每行多少个字节
```

该函数使用 z 缓存绘制使用 Gouraud 着色的三角形，其效果比前一个版本更好，因为它的精度超过像素级，且遵守了左上填充规则。调用方法与前一版本相同。

```
void Draw_RENDERLIST4DV2_SolidZB2_16(RENDERLIST4DV2_PTR rend_list,
                UCHAR *video_buffer,
              int lpitch,
                UCHAR *zbuffer,
                int zpitch);
```

该函数是调用上述函数（使用 z 缓存的光栅化函数）的入口。除了不再执行截尾操作外，其代码与前一个版本相同。

在前一章结束时，我们能够绘制使用固定、恒定或 Gouraud 着色的实心多边形，还能绘制使用仿射纹理映射和固定或恒定着色的多边形。前面介绍的函数也完成这些功能，但精度更高，误差更小。这里不提供使用这些函数的演示程序——它们的效果与以前的函数差别不大。接下来介绍一些效果截然不同的技术。

12.3　使用 Gouraud 着色的纹理映射

这是第一个支持使用 Gouraud 着色的纹理映射的光栅化库版本。作者最初以为使用软件来实现它的速度会很慢，但试验后发现速度足够快。使用定点数的光栅化函数确实很快，为支持 Gouraud 着色需要执行更多的插值运算，这使得内循环更为复杂，但速度并没有降低很多。为支持使用 Gouraud 着色的纹理映射，需要做哪些工作呢？图 12.1 说明了使用固定着色的仿射纹理映射和单色 Gouraud 着色。窍门是将它们合而为一。

我们的进攻计划很简单：以绘制单色多边形的 Gouraud 着色函数为基础，加入通过插值计算纹理坐标 u 和 v 的代码。下面是 Gouraud 着色函数（非 z 缓存版本）的内循环代码：

```
// 将指针 screen_ptr 指向起始行
screen_ptr = dest_buffer + (ystart * mem_pitch);

for (yi = ystart; yi < yend; yi++)
   {
   // 计算扫描线端点的 x 坐标
   xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
   xend   = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

   // 计算 u、v 和 w 的起始值
   ui = ul + FIXP16_ROUND_UP;
   vi = vl + FIXP16_ROUND_UP;
   wi = wl + FIXP16_ROUND_UP;
```

```
 // 计算 u、v 和 w 的增量
 if ((dx = (xend - xstart))>0)
{
du = (ur - ul)/dx;
dv = (vr - vl)/dx;
dw = (wr - wl)/dx;
} // end if
else
{
du = (ur - ul);
dv = (vr - vl);
dw = (wr - wl);
} // end else

// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
{
// 绘制像素
  screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
         ((vi >> (FIXP16_SHIFT+2)) << 5) +
         (wi >> (FIXP16_SHIFT+3));

// 计算下一个像素的 u、v 和 w
ui+=du;
vi+=dv;
wi+=dw;
} // end for xi

  // 计算下一条扫描线端点的 u、v、w 和 x
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;
wl+=dwdyl;

xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
wr+=dwdyr;

// 将指针 screen_ptr 指向下一行
screen_ptr+=mem_pitch;
} // end for y
```

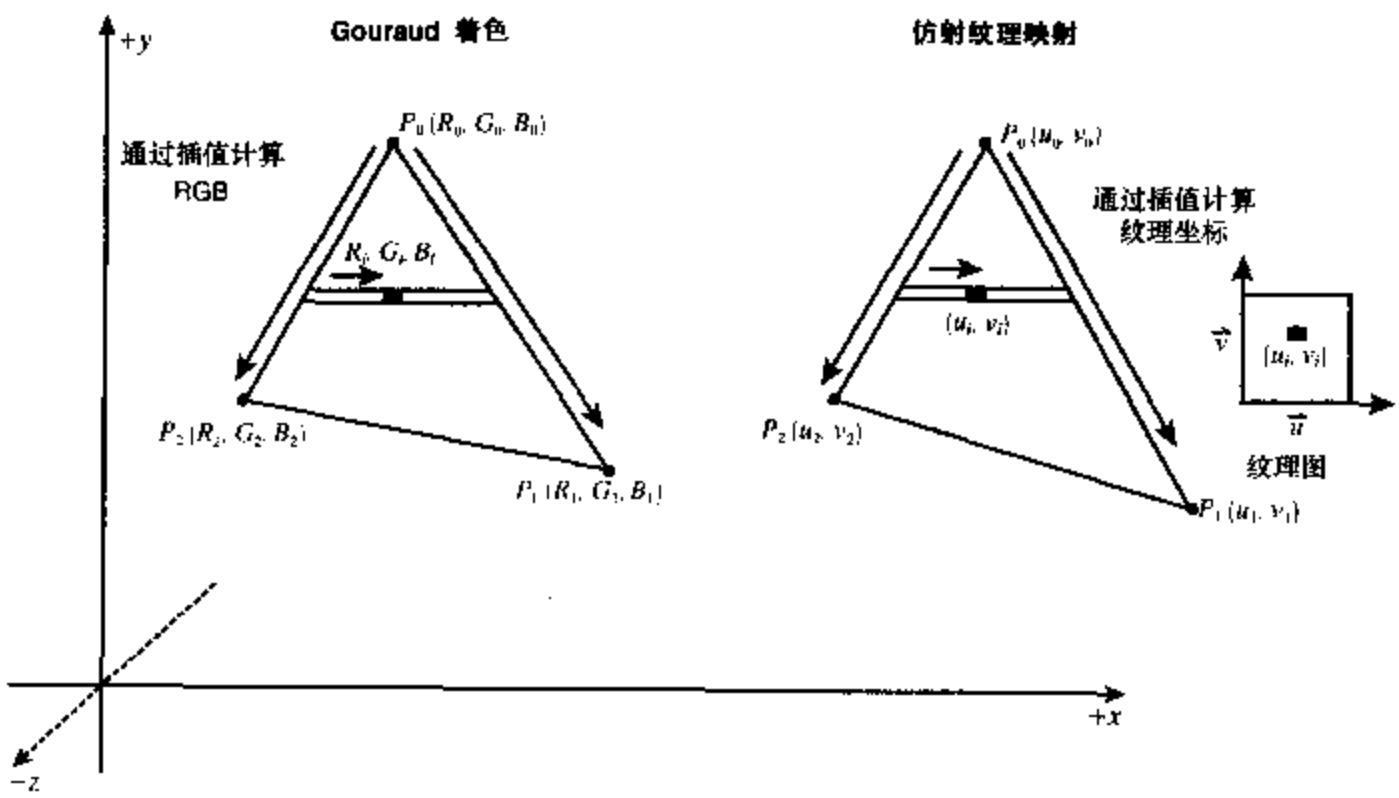


图 12.1　将使用固定着色的纹理映射和 Gouraud 着色合而为一

其中最内面的循环代码（绘制一条扫描线）为粗体，它使用 16.16 定点格式，通过插值计算 R、G、B 分量，并将其存储在变量 u、v 和 w 中。下面的代码段根据定点值生成一个 5.6.5 格式的 16 位 RGB 字，并将其绘制到屏幕上：

```
screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
        ((vi >> (FIXP16_SHIFT+2)) << 5) +
        (wi >> (FIXP16_SHIFT+3));
```

除此之外，Gouraud Shader 没有其他特别之处。下面来看看仿射纹理映射函数中的插值代码：

```
// 将指针 screen_ptr 指向起始行
screen_ptr = dest_buffer + (ystart * mem_pitch);

for (yi = ystart; yi < yend; yi++)
  {
  // 计算扫描线端点的 x 坐标
  xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
  xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

  // 计算 u 和 v 的起始值
  ui = ul + FIXP16_ROUND_UP;
  vi = vl + FIXP16_ROUND_UP;

  // 计算 u 和 V 的增量
  if ((dx = (xend - xstart))>0)
    {
  du = (ur - ul)/dx;
  dv = (vr - vl)/dx;
  } // end if
  else
    {
    du = (ur - ul);
    dv = (vr - vl);
  } // end else

  // 绘制扫描线
  for (xi=xstart; xi < xend; xi++)
  {
  // 绘制像素
    screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
                ((vi >> FIXP16_SHIFT) << texture_shift2)];

  // 计算下一个像素的 u 和 v
  ui+=du;
  vi+=dv;
  } // end for xi

  // 计算下一条扫描线端点的 u、v 和 x
  xl+=dxdyl;
  ul+=dudyl;
  vl+=dvdyl;

  xr+=dxdyr;
  ur+=dudyr;
  vr+=dvdyr;

  // 将指针 screen_ptr 指向下一行
  screen_ptr+=mem_pitch;
  } // end for y
```

同样，绘制像素的代码也很简单。将定点数纹理坐标 ui 和 vi 从 16.16 定点格式转换为整数，然后使用它们来访问纹理图，再将纹素写入到内存中，仅此而已。因此，将这两个函数合并成一个不应太难。读者需要牢记的是，我们使用 Gouraud 着色来处理纹理，但和使用恒定着色的纹理映射函数中一样，我们假定纹理下面的多边形最初的颜色为白色，据此执行光照计算，然后使用光照处理后的多边形颜色来调制纹理颜色。图 12.2 说明了这个处理过程中的数据流程。

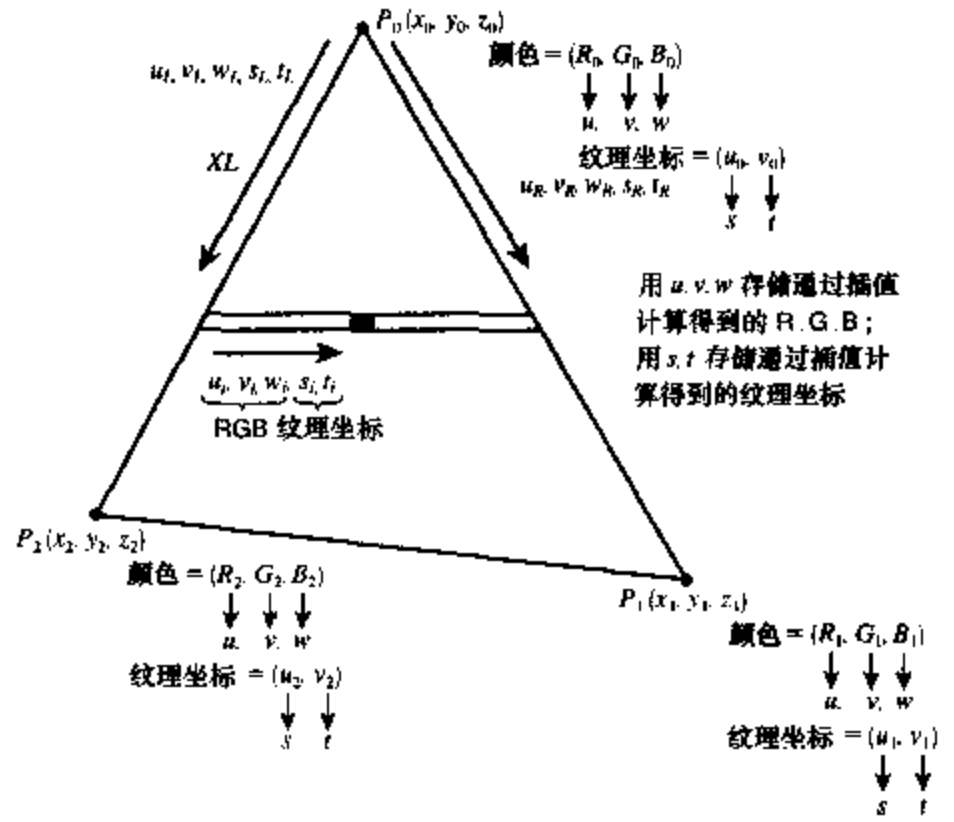


图 12.2　Gouraud 纹理映射函数中的插值计算流程

基本上，同时执行 Gouraud 着色和纹理映射。Gouraud 着色部分通过插值计算 R、G、B（用变量 u、v、w 表示），而纹理映射部分通过插值计算 u 和 v（用变量 s 和 t 表示）。最后是存取纹素和颜色调制。下面是从完成后的 Gouraud 纹理映射函数中摘录的代码：

```
// 将指针 screen_ptr 指向起始行
screen_ptr = dest_buffer + (ystart * mem_pitch);

for (yi = ystart; yi < yend; yi++)
  {
// 计算扫描线端点的 x 坐标
xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

// 计算扫描线起点的 u、v、w、s 和 t
ui = ul + FIXP16_ROUND_UP;
vi = vl + FIXP16_ROUND_UP;
wi = wl + FIXP16_ROUND_UP;

si = sl + FIXP16_ROUND_UP;
ti = tl + FIXP16_ROUND_UP;
// 计算 u 和 v 的增量
if ((dx = (xend - xstart))>0)
{
du = (ur - ul)/dx;
dv = (vr - vl)/dx;
dw = (wr - wl)/dx;

ds = (sr - sl)/dx;
dt = (tr - tl)/dx;
} // end if
else
{
du = (ur - ul);
dv = (vr - vl);
dw = (wr - wl);

ds = (sr - sl);
dt = (tr - tl);

} // end else

// 绘制扫描线
```

```
for (xi=xstart; xi < xend; xi++)
{
  // 首先读取纹素
  textel = textmap[(si >> FIXP16_SHIFT) +
          ((ti >> FIXP16_SHIFT) << texture_shift2)];

  // 提取 R、G、B 分量
  r_textel = ((textel >> 11)    );
  g_textel = ((textel >> 5) & 0x3f);
  b_textel =  (textel    & 0x1f);

  // 使用 Gouraud 着色得到的颜色对纹素进行调制
  r_textel*=ui;
  g_textel*=vi;
  b_textel*=wi;

  // 最后绘制像素
  screen_ptr[xi] = ((b_textel >> (FIXP16_SHIFT+8)) +
          ((g_textel >> (FIXP16_SHIFT+8)) << 5) +
          ((r_textel >> (FIXP16_SHIFT+8)) << 11));

// 计算下一个像素的 u、v、w、s、t
ui+=du;
vi+=dv;
wi+=dw;

si+=ds;
ti+=dt;

} // end for xi

// 计算下一条扫描线起点的 u、v、w、x、s 和 t
xl+=dxdyl;
ul+=dudyl;
vl+=dvdyl;
wl+=dwdyl;

sl+=dsdyl;
tl+=dtdyl;


xr+=dxdyr;
ur+=dudyr;
vr+=dvdyr;
wr+=dwdyr;

sr+=dsdyr;
tr+=dtdyr;

// 将指针 screen_ptr 指向下一行
screen_ptr+=mem_pitch;

} // end for y
```

和以往一样，内循环代码用粗体显示。它首先提取要对其进行 Gouraud 着色的纹素：

```
r_textel = ((textel >> 11)    );
g_textel = ((textel >> 5) & 0x3f);
b_textel =  (textel    & 0x1f);
```

现在，r_textel、g_textel 和 b_textel 中存储的是以 5.6.5 格式表示的纹素的 RGB 值。需要将它们分别乘以对多边形进行 Gouraud 着色得到的颜色的 R、G、B 分量，代码如下：

```
r_textel*=ui;
g_textel*=vi;
b_textel*=wi;
```

现在暂停一会儿，来看看定点数运算情况。进行颜色调制之前，r_textel、g_textel 和 b_textel 不是定点

数，而是整数，它们的取值范围分别是 0～31、0～63 和 0～31。然而，ui、vi 和 wi 是定点数，将定点数和整数相乘时，结果为定点数。现在，r_textel、g_textel 和 b_textel 存储的是对纹理进行 Gouraud 着色后得到的颜色的 R、G、B 分量，且为定点数，因此需要从这三个变量中提取正确的数据。为此，需要对它们进行缩放和移位，然后通过组合得到最终的像素颜色，然后绘制它。下面的代码通过位运算来进行缩放和移位，并将 R、G、B 分量格式化为 5.6.5 格式，然后绘制到屏幕上：

```
screen_ptr[xi] = ((b_textel >> (FIXP16_SHIFT+8)) +
        ((g_textel >> (FIXP16_SHIFT+8)) << 5) +
        ((r_textel >> (FIXP16_SHIFT+8)) << 11));
```

注意：例如，将定点数 10 和 10 相加时，结果为定点数 20，这与将定点数 10 乘以 2 等效。下面来看看其原因。使用 16.16 格式时，定点数 10 为 10*65536（这是因为它将被存储在前 16 位中，这相当于左移 16 位，即乘以 2^{16}——译者注）。因此，将两个定点数 10 相加时，结果为：

```
(10*65536) + (10*65536) = (20*65536) = (2*10*65536) = 2*(10*65536)
```

因此，将定点数乘以或除以整数时，结果仍为定点数。

Gouraud 纹理映射函数进行光栅化时，对于每个像素，只需执行三次乘法运算和几次加法/移位运算，因此速度非常快。这就是作者坚持使用定点数的原因——如果在这个函数中使用浮点数，浮点数转换的开销将让您承担不起。

读者可能在考虑提高该函数的速度的方法。方法有很多，但就算法而言，该函数的代码几乎是最优的，这意味着为完成工作，它所做的工作量几乎是最少的。可以采用两种位缩放（bit-scaling）操作，但从很大程度上说，要得到相同的结果，几乎没有办法再减少工作量了。

使用查找表可能会有所帮助，但它们会带来存储空间的问题，我不能确定是否有这么多的内存供它们使用。最明显的方法是，创建一个查找表，在其中存储了各种纹素颜色和 Gouraud 插值计算得到的各种 RGB 值的乘积。这个表将非常大，因为需要包含两个 5.6.5 格式 RGB 值的全部组合，一共有 65536*665536 = 2^{32} 组合，大约是 40 亿。这太大了。另一种方法是，压缩颜色空间，例如将其压缩到 12 位，这将有 2^{12}*2^{12} 种组合，大约是 1670 万。这仍然太大，而颜色空间却缩小了很多。并非不能使用查找表，但可能是不明智的。要提高上述代码的速度，只能重新调整顺序和进行其他微妙的修改——其结构已经非常好了。

下面是不使用 z 缓存的 Gouraud 纹理映射函数的原型：

```
void Draw_Textured_TriangleGS_16(POLYF4DV2_PTR face,  // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch)     // 每行多少个字节（320、640 等）
```

其调用方法与恒定着色版本相同，但由于它使用 Gouraud 着色和纹理映射，因此您必须设置纹理坐标 u 和 v 以及顶点颜色，后者必须存储在顶点颜色数组 lit_color[ ]中。下面是该函数的 z 缓存版本：

```
void Draw_Textured_TriangleGSZB_16(POLYF4DV2_PTR face,// 指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少个字节（320、640 等）
            UCHAR *_zbuffer,   // 指向 z 缓存的指针
            int zpitch);     // z 缓存中每行多少个字节
```

同样调用该函数时，必须设置多边形的纹理坐标和顶点颜色，另外还需要指定 z 缓存及其跨距。例如，下面的代码摘自一个渲染函数，它们为调用 Gouraud 纹理映射函数做准备：

```
// 设置顶点坐标和纹理坐标
face.tvlist[0].x = (float)rend_list->poly_ptrs[poly]->tvlist[0].x;
face.tvlist[0].y = (float)rend_list->poly_ptrs[poly]->tvlist[0].y;
face.tvlist[0].z = (float)rend_list->poly_ptrs[poly]->tvlist[0].z;
face.tvlist[0].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[0].u0;
face.tvlist[0].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[0].v0;

face.tvlist[1].x = (float)rend_list->poly_ptrs[poly]->tvlist[1].x;
face.tvlist[1].y = (float)rend_list->poly_ptrs[poly]->tvlist[1].y;
face.tvlist[1].z = (float)rend_list->poly_ptrs[poly]->tvlist[1].z;
face.tvlist[1].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[1].u0;
```

```
face.tvlist[1].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[1].v0;

face.tvlist[2].x = (float)rend_list->poly_ptrs[poly]->tvlist[2].x;
face.tvlist[2].y = (float)rend_list->poly_ptrs[poly]->tvlist[2].y;
face.tvlist[2].z = (float)rend_list->poly_ptrs[poly]->tvlist[2].z;
face.tvlist[2].u0 = (float)rend_list->poly_ptrs[poly]->tvlist[2].u0;
face.tvlist[2].v0 = (float)rend_list->poly_ptrs[poly]->tvlist[2].v0;

// 指定纹理
face.texture = rend_list->poly_ptrs[poly]->texture;

// 设置顶点颜色
face.lit_color[0] = rend_list->poly_ptrs[poly]->lit_color[0];
face.lit_color[1] = rend_list->poly_ptrs[poly]->lit_color[1];
face.lit_color[2] = rend_list->poly_ptrs[poly]->lit_color[2];
```

由于 Gouraud 着色和纹理映射都介绍过多次，因此这里介绍 Gouraud 纹理映射的步伐非常快，我们只是将这两种技术合并到一个函数中而已。

有关使用 Gouraud 纹理映射的演示程序，请参阅 DEMOII12_1.CPP|EXE，图 12.3 是该演示程序运行时的屏幕截图。它在屏幕上显示一组从左到右分别使用固定、恒定和 Gouraud 着色且都带纹理的物体，让读者能够看到质量有何不同。要更换物体，可按 1、2 和 3 键。对于带角的物体，Gouraud 纹理映射并没有多大帮助，但对于圆润的物体，它确实有很大的影响。

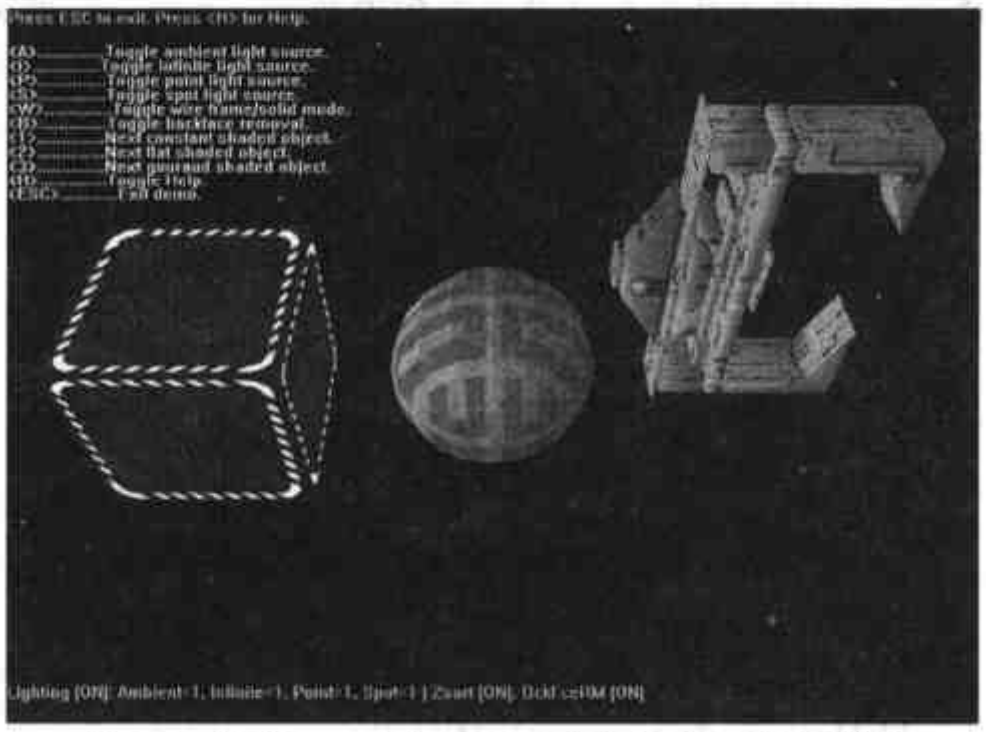


图 12.3　使用 Gouraud 纹理映射函数的演示程序的屏幕截图

12.4　透明度和 alpha 混合

说实话，对于在引擎中加入透明度支持会不会极大地降低速度，作者有些担心。从理论上说，alpha 混合/透明度只不过是简单的混合操作。给定两个像素 p_source1(r，g，b)和 p_source2(r，g，b)和混合因子（我们将其称为 alpha，其取值范围为 0～1），计算最终像素 p_final(r，g，b)（将前两个像素混合起来）的公式如下：

```
p_final = (alpha)*(p_source1) + (1-alpha)*(p_source2)
```

每个像素实际上都由三个分量 r、g 和 b 组成，因此上述公式实际上是这样的：

```
p_finalr = (alpha)*(p_source1r) + (1-alpha)*(p_source2r)
p_finalg = (alpha)*(p_source1g) + (1-alpha)*(p_source2g)
p_finalb = (alpha)*(p_source1b) + (1-alpha)*(p_source2b)
```

如果 alpha = 1，则 p_source1 是完全不透明的，p_source2 无论是什么样的都无关紧要。同样，如果 alpha = 0，

则p_source1是完全透明的，而p_source2是完全不透明的。最后，如果alpha = 0.5，将按50/50的方式混合p_source1和p_source2。这就是透明度/alpha混合技术的基础。我们面临的主要问题有三个：

- 如何能够以足够快的速度执行这种复杂的数学运算？
- 如何在引擎中支持alpha混合？即如何指出多边形有alpha值？
- 如何处理DirectX在读取后缓存时速度将降低30倍的问题？

后两个问题将推迟到后面去解决，因为它们是更为高级的问题；现在只解决第一个问题。

对于每个像素，alpha混合都涉及6次乘法运算和3次加法运算。在大量使用alpha混合的情况下，仅这些运算就让我们承担不起，因此蛮干（brute force）是绝对行不通的。我们确实可以在光栅化循环中对每个像素执行alpha混合运算，但这是不能接受的。有更好的办法，这就是查找表。

12.4.1 使用查找表来进行alpha混合

我们面临的问题时，为支持alpha混合，计算量很大，如下所示：

```
p_finalr = (alpha)*(p_source1r) + (1-alpha)*(p_source2r)
p_finalg = (alpha)*(p_source1g) + (1-alpha)*(p_source2g)
p_finalb = (alpha)*(p_source1b) + (1-alpha)*(p_source2b)
```

这包括6次乘法运算和4次加法运算（其中一次是计算常量1-alpha）。这不仅仅是数学运算的问题：除了对像素执行上述计算外，还需要提取颜色分量值，因为它们是以RGB格式存储的。因此，我们将创建一个这样的查找表：通过它可以根据两个5.6.5格式的RGB值和一个0～255的alpha值得到混合结果，如图12.4所示。当然，我们再次面临困境：查找表非常大。下面来计算查找表的大小。

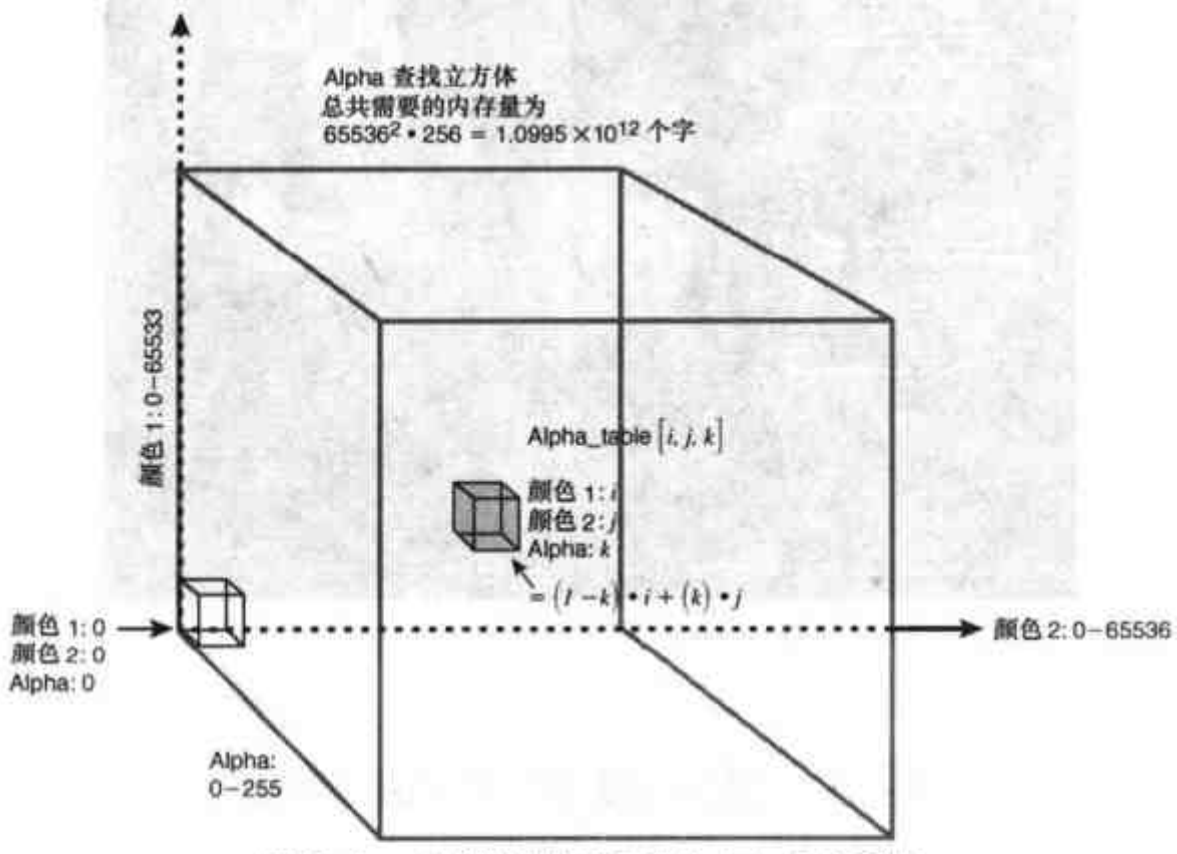


图12.4 一种使用查找表实现的alpha混合系统

查找表的输入有三个：

- 颜色1：RGB格式，长16位；
- 颜色2：RGB格式，长16位；
- 混合因子：0～255，长8位。

因此，查找表需要存储颜色1、颜色2和混合因子的各种组合的结果，总组合数为：

$$65536 * 65536 * 256 = 1.0995\times10^{12}$$

不用说，如果不进行调整，这种方法是不可行的。下面从另一个角度来考虑这个问题。再来看看alpha混合的计算公式：

```
p_finalr = (alpha)*(p_source1r) + (1-alpha)*(p_source2r)
```

```
p_finalg = (alpha)*(p_source1g) + (1-alpha)*(p_source2g)
p_finalb = (alpha)*(p_source1b) + (1-alpha)*(p_source2b)
```

如果不考虑一次性解决整个问题，可以将重点放在子问题上。在上述公式中，主要的计算是将 alpha 因子以及 1-alpha 分别与 R、G、B 分量相乘。我们可以创建一个包含 65536 个条目的查找表，每个条目都是特定 alpha 值（如 0.5）和不同 5.6.5 格式的 RGB 值相乘的结果，如图 12.5 所示。

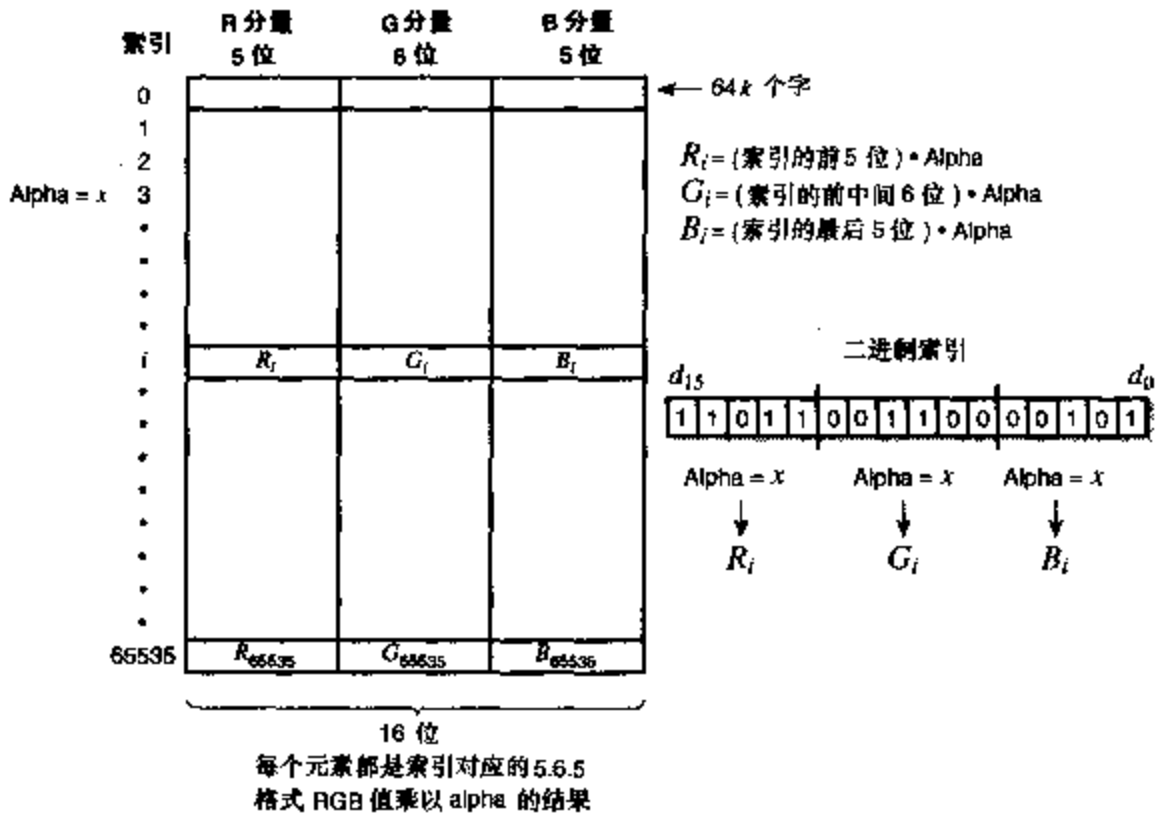


图 12.5　一个效率更高的 alpha 查找表系统

有了上述查找表后，要计算 alpha 为 0.5 时两个像素的混合结果，只需执行两次表查找操作和一个加法运算，如下所示：

```
p_final = alphatable[p_source1] + alphatable[p_source2];
```

通过使用一个包含 65536 个条目，每个条目占用 16 位的查找表，我们将下述计算：

```
p_finalr = (alpha)*(p_source1r) + (1-alpha)*(p_source2r)
p_finalg = (alpha)*(p_source1g) + (1-alpha)*(p_source2g)
p_finalb = (alpha)*(p_source1b) + (1-alpha)*(p_source2b)
```

转换为了如下运算：

```
p_final = alphatable[p_source1] + alphatable[p_source2];
```

这种运算的速度非常快。然而，您可能需要使用其他的 alpha 值。每个 alpha 查找表的大小为 64K 个字，如果要下面支持 8 种均匀分布的 alpha 值：

```
0, .125, .250, .375, .5, .625, .75, 1.00
```

将需要 8 个 alpha 查找表，总共为 8*64 = 521KB 个字。这是能够承担得起的。

注意：读者可能注意到了，并不需要 alpha 为 0 和 alpha 为 1.0 的查找表。我们可以使用条件逻辑来处理这两种情况：如果 alpha 为 0，则像素是完全透明的，只需要将另一个像素作为计算结果即可；同样，如果 alpha 为 1.0，只需向通常那样绘制像素即可，根本不需要考虑 alpha 混合，因为 alpha 为 1.0 表示完全不透明。就现在而言，为使支持 alpha 混合的光栅化函数更为清晰，我们将使用这两个 alpha 查找表，而不考虑这两种特殊情形。

最后，我们将为一组 0～1 的 alpha 值创建 alpha 查找表，在其中存储 alpha 值和各种 RGB 值的乘积。计算乘积时，我们不计算 alpha 值和 0～65535 的每个整数的乘积，而是从整数中提取 R、G、B 分量，然后将 alpha 值分别同它们相乘。这种算法的伪代码如下：

```
for(0 到 65535 的每个颜色值)
```

```
begin
  将颜色值视为 5.6.5 格式，从中提取 R、G、B 分量

  然后将每个分量与 alpha 值相乘

  r=r*alpha
  g=g*alpha
  b=b*alpha
  最后根据 R、G、B 分量生成一个 16 位的颜色字
  并其存储到查找表中与颜色值对应的位置：

  alphatable[value] = r.g.b

end
```

alpha 查找表很容易创建。关键之处在于，16 位的索引表示要将其乘以 alpha 值的颜色，但不是将索引与 alpha 值相乘，而是从索引中提取 R、G、B 分量，然后将它们分别乘以 alpha 值，再将它们组合成一个 16 位颜色值。为支持足够多的 alpha 值以便能够正确地进行混合，需要创建多个 alpha 查找表，可能是 8 个或 6 个。作者决定使用 8 个 alpha 查找表，但读者可以通过修改常量来控制 alpha 查找表数目。下面是存储 alpha 查找表的全局数组，它是在 T3DLIB10.CPP 中声明的：

```
USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536];
```

当前 NUM_ALPHA_LEVEL 被设置为 8，支持 8 个 alpha 等级，每个等级之间相差 0.125。另外，正如读者看到的，alpha 查找表是按行分布的，每行表示一个不同的 alpha 值，图 12.6 说明了这一点。

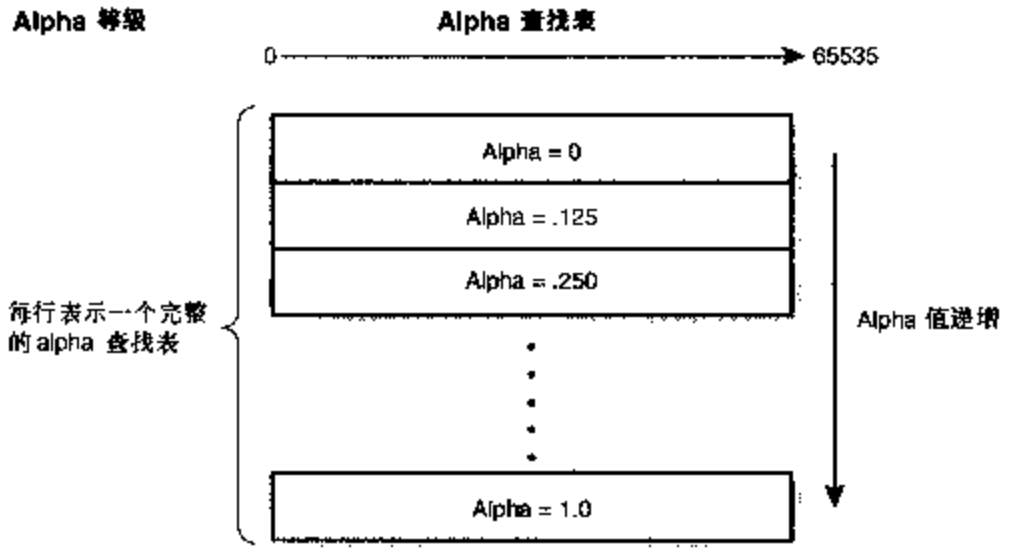


图 12.6　使用多个查找表

下面是用于创建 alpha 查找表的函数 RGB_Alpha_Table_Builder()：

```
int RGB_Alpha_Table_Builder(int num_alpha_levels, // alpha 等级数
USHORT rgb_alpha_table[NUM_ALPHA_LEVELS][65536]) // 查找表
{
// 该函数创建 alpha 查找表
// 创建该查找表的原因如下：
// 不能使用下述公式来对两个 RGB 值进行 alpha 混合
// final = (alpha)*(source1RGB) + (1-alpha)*(source2RGB)
// 原因是需要分别对 R、G、B 分量执行乘法运算
// 这要求首先从源像素中提取 R、G、B，将其分别与 alpha 和 1-alpha 相乘
// 再将结果相加得到最终的像素值
// 为提高上述运算的速度，可使用一个查找表
// 如果将输入的 RGB 视为一个数字，将其与 alpha 因子相乘的结果存储在查找表中
// 便可以将 alpha 混合运算简化为两次查表操作和一次加法运算
// 从而避免 6 次乘法运算和所有的移位运算
// 这个函数创建一个包含 num_alpha_levels 行的查找表
// 每个包含 65536 个条目
// 各个条目为 RGB 值与 alpha 因子的乘积
// 另外，num_alpha_levels 必须是 2 的幂
// 如果 num_alpha_levels 为 8，查找表将包含 8 行
// 每行分别是 65536 种 RGB 值的 8/8、7/8、6/8...1/8、0/8
```

```
// 检查指针是否有效
if (!rgb_alpha_table)
  return(-1);

int r,g,b; // 用于存储从 rgbindex 中提取的 R、G、B 分量

float alpha    = 0;
float delta_alpha = EPSILON_E6 + 1/((float)(num_alpha_levels-1));

// 需要 num_alpha_level 行
for (int alpha_level = 0; alpha_level < num_alpha_levels; alpha_level++)
  {
// 假设使用的是 RGB 4.4.4 格式
// 有 65536 个 RGB 需要计算
for (int rgbindex = 0; rgbindex < 65536; rgbindex++)
  {
  // 假设格式为 5.6.5，从 rgbindex 中提取 R、G、B 分量
  _RGB565FROM16BIT(rgbindex, &r, &g, &b);

  // 计算乘积
  r = (int)( (float)r * (float)alpha );
  g = (int)( (float)g * (float)alpha );
  b = (int)( (float)b * (float)alpha );

  // 生成 RGB 值，并将其存储到查找表中
  rgb_alpha_table[alpha_level][rgbindex] = _RGB16BIT565(r,g,b);

  } // end for rgbindex

 // 计算下一个 alpha 值
  alpha+=delta_alpha;

 } // end for row

// 成功返回
return(1);

} // end RGB_Alpha_Table_Builder
```

要使用该函数，只需使用 alpha 等级数和存储 alpha 查找表的数组作为参数调用它即可。在本章中，它们是常量和全局变量，因此这样调用该函数：

```
RGB_Alpha_Table_Builder(NUM_ALPHA_LEVELS, rgb_alpha_table);
```

创建 alpha 查找表后，接下来介绍如何使用它们。

1. 在基本光栅化函数中加入 alpha 混合功能

将 alpha 混合功能加入到基本的非 z 缓存光栅化函数中最为容易，因此从这里开始。首先来回顾一下为何要使用 alpha 混合。使用 alpha 混合旨在透过前面的东西看到后面的东西，如图 12.7 所示。

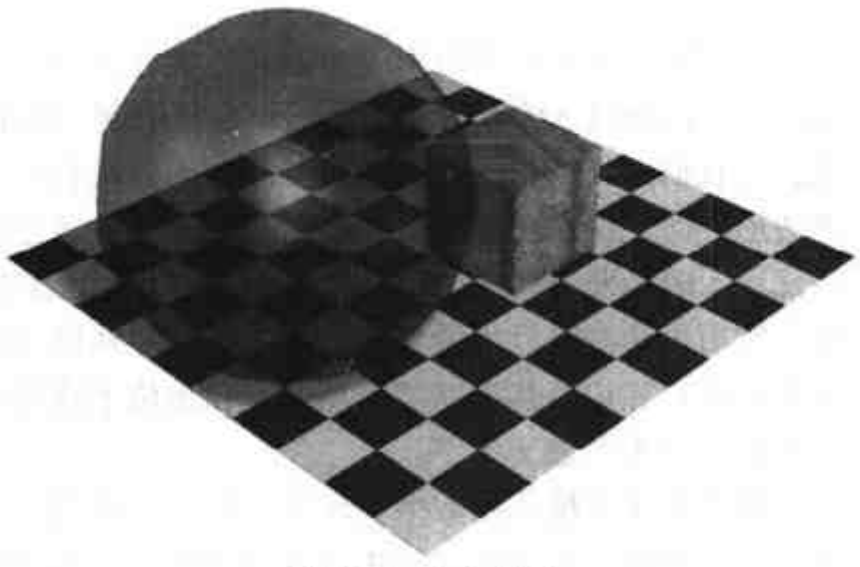


图 12.7　alpha 混合

图中有一个场景，然后在场景上渲染了一个物体，其 alpha 值大约为 20%。这样，背景占 80%，前景占 20%。这提出了第一个与 alpha 混合和透明性有关的问题：必须先绘制位于透明物体后面的物体。这是很多算法都面临的一个问题。例如，我们可能有一个非常好的算法，它不会重复绘制，即不绘制被其

他多边形遮掩的多边形，如图 12.8 所示。问题是，如果前面的多边形是透明的，透过它将看不到任何东西。

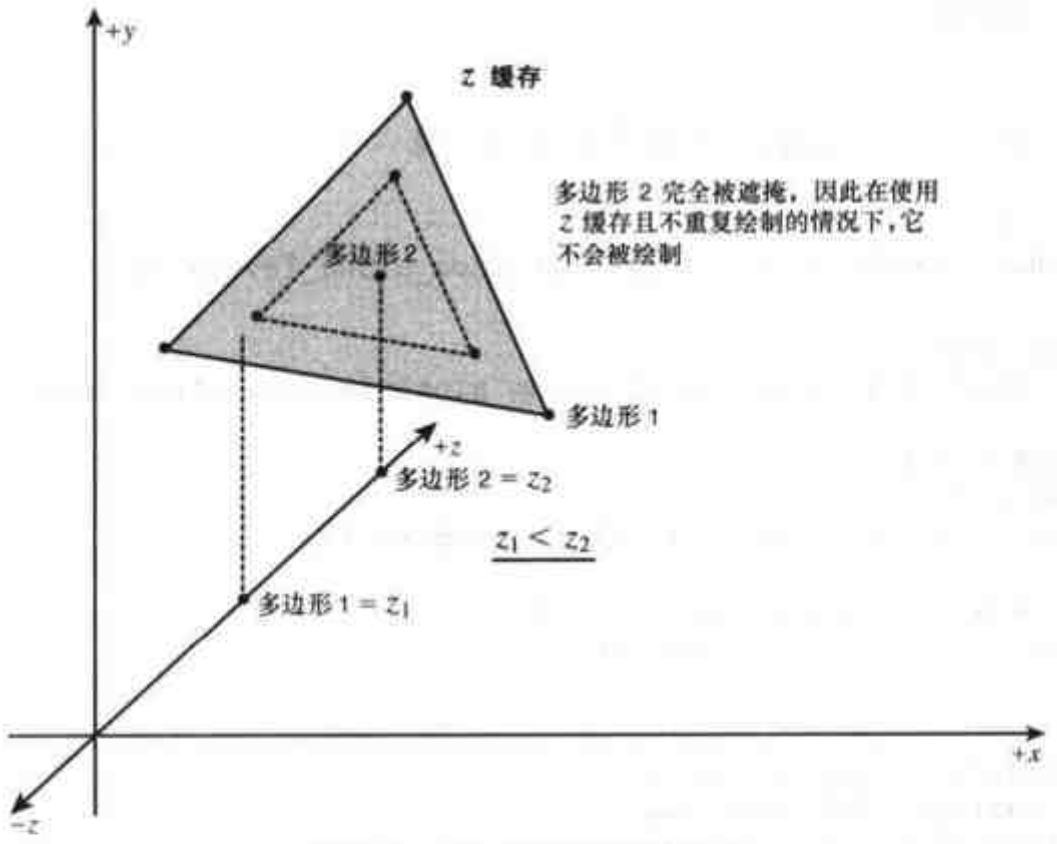


图 12.8　不绘制被遮掩的多边形

另一个问题出现在使用 *z* 缓存时。假设两个多边形按从前到后的顺序经过 *z* 缓存测试，即多边形 1 离视点的距离比多边形 2 近，如图 12.9 所示。

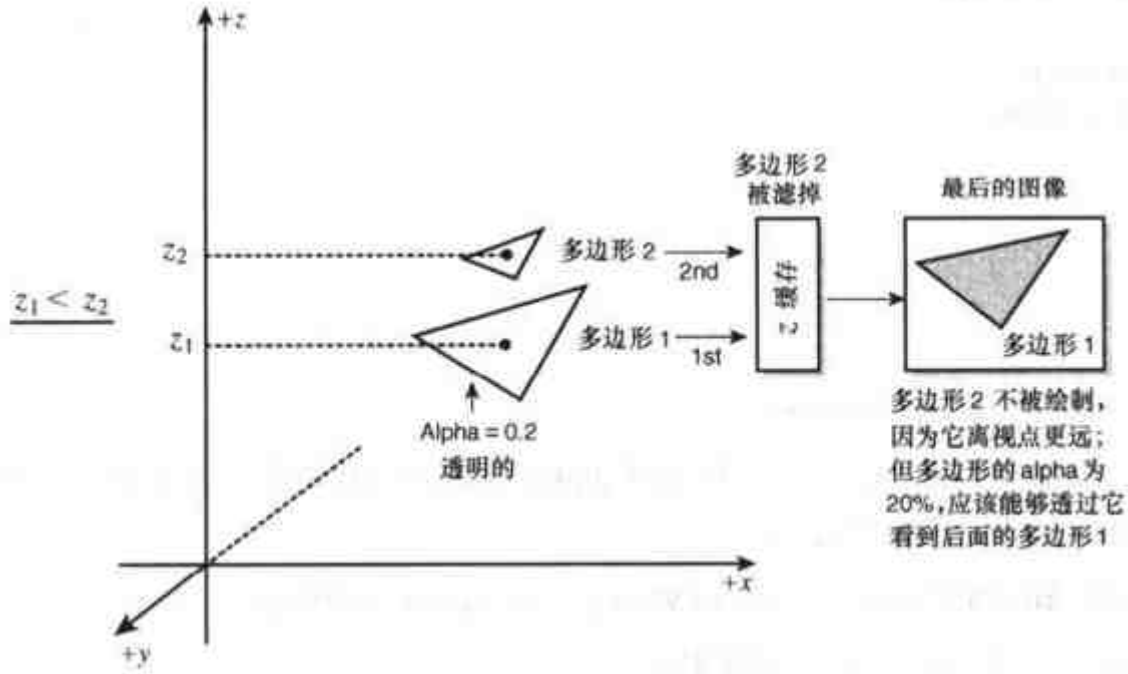


图 12.9　使用 *z* 缓存时 alpha 混合存在的问题

在这种情况下，多边形 1 将通过 *z* 缓存测试，进而被绘制到屏幕上，其 z 值被写入到 *z* 缓存中。但多边形 2 不能通过 *z* 缓存测试，因此不会被绘制。然而，多边形 1 是透明的，应该能够透过它看到多边形 2。这就是我们面临的困境。解决这些问题的方法有很多。例如，在试图避免重复绘制时，可以不考虑透明的多边形，即无论情况如何都绘制它们。至于 *z* 缓存的问题，可以按从后到前的顺序大概地排列多边形，然后使用 *z* 缓存来实现绝对正确的顺序，并在绘制透明多边形时进行 alpha 混合；这样将首先绘制较远的多边形，然后再绘制较近的透明多边形，以便能够进行混合。如果不想浪费时间来给多边形排序，可以对透明多边形进行标记，然后在渲染期间，找出位于透明多边形后面且完全位于其包围框内的多边形，并首先绘制它们，而不考虑 z 顺序。

接下来在光栅化函数中实现透明性，从最简单的光栅化函数开始，它绘制使用固定或恒定着色处理的多边形，不使用 *z* 缓存，也不进行纹理映射。这个新函数名为 Draw_Triangle_2D3_16()。为支持 alpha 混合，可给函数增加一个参数，用于指定 alpha 等级，并在函数的内循环中加入执行 alpha 混合的代码。首先

再来看一下内循环的代码，以制定更新计划：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  // 绘制像素
  screen_ptr[xi] = color;
  } // end for xi
```

为支持透明性，需要读取当前的像素 screen_ptr[xi]，将其同 color 进行混合，而不是仅仅将 color 存储到 screen_ptr[xi]中。我们将使用强大的查找表来完成这项工作。假设查找表存储在全局数组中，这样无需在调用函数时传递它；再假设调用光栅化函数时，将传递一个 0 到 NUM_ALPHA_LEVELS – 1 的 alpha 等级（参数 alpha），然后将 alpha 作为索引，让一个指针指向 alpha 查找表数组的第 alpha 行，让另一个指针指向第（NUM_ALPHA_LEVELS – alhpa – 1）行。例如，如果 alpha 等级为 0，则一个指针指向第 0 行，另一个指针指向（NUM_ALPHA_LEVELS – 1）行。设置指针的代码如下：

```
// 根据多边形的 alpha 等级确定要使用的查找表
USHORT *alpha_table_src1 =
    (USHORT *)&rgb_alpha_table[(NUM_ALPHA_LEVELS-1) - alpha][0];
USHORT *alpha_table_src2 =

    (USHORT *)&rgb_alpha_table[alpha][0];
```

现在，只需将屏幕上的像素和要渲染到屏幕上的像素混合起来即可。为此，只需使用利用查找表的混合算法。新的内循环代码如下：

```
for (xi=xstart; xi < xend; xi++)
{
// 绘制像素
screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] + alpha_table_src2[color];
} // end for xi
```

我们将复杂的 alpha 混合问题简化成了两次表查找和一次加法运算。现在，读者应该知道如何在每个光栅化函数中加入 alpha 混合功能：只需根据 alpha 等级选择 alpha 查找表，然后读取屏幕上的像素，并将其与要渲染的像素/纹素混合起来即可。

（1）在支持简单着色的光栅化函数中加入 alpha 混合功能

支持 alpha 混合、不使用 z 缓存、对使用固定/恒定着色的多边形进行光栅化的函数名为 Draw_Triangle_2D_Alpha16()，其原型如下：

```
void Draw_Triangle_2D_Alpha16(POLYF4DV2_PTR face, // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少个字节(320、640 等)
          int alpha);      // alpha 等级
```

除了新增参数 alpha 外，该函数的原型与非 alpha 版相同。参数 alpha 必须为 0 到 NUM_ALPHA_LEVELS – 1；否则访问 alpha 查找表数组时将越界，导致内存保护错误。接下来在支持纹理映射和 Gouraud 着色的基本光栅化函数中加入 alpha 混合功能。

（2）在 Gouraud 着色中加入 alpha 混合功能

这很简单，只需使用下述代码行根据 alpha 等级选择 alpha 查找表：

```
// 根据多边形的 alpha 等级确定要使用的查找表
USHORT *alpha_table_src1 =
   (USHORT *)&rgb_alpha_table[(NUM_ALPHA_LEVELS-1) - alpha][0];
USHORT *alpha_table_src2 =

    (USHORT *)&rgb_alpha_table[alpha][0];
```

然后将下述内循环：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
```

```
// 绘制像素
screen_ptr[xi] = ((ui >> (FIXP16_SHIFT+3)) << 11) +
         ((vi >> (FIXP16_SHIFT+2)) << 5) +
         (wi >> (FIXP16_SHIFT+3));

// 计算下一个像素的 u、v、w
ui+=du;
vi+=dv;
wi+=dw;
} // end for xi
```

修改成这样即可：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  // 绘制像素
  screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] +
          alpha_table_src2[((ui >> (FIXP16_SHIFT+3)) << 11) +
                 ((vi >> (FIXP16_SHIFT+2)) << 5) +
                 (wi >> (FIXP16_SHIFT+3))];

  // 计算下一个像素的 u、v、w
  ui+=du;
  vi+=dv;
  wi+=dw;
  } // end for xi
```

现在，原来绘制像素的代码被用作索引，用于在 alpha 查找表中查找屏幕像素的贡献。因此，对这种函数进行更新非常简单：将原来绘制像素的代码用作一个 alpha 查找表的索引，并把要将值写入其中的像素地址用作写入目标和另一个 alpha 查找表的索引。下面是支持 alpha 混合的 Gouraud 着色函数的原型：

```
void Draw_Gouraud_Triangle_Alpha16(POLYF4DV2_PTR face,// 指向多边形面的指针
           UCHAR *_dest_buffer, // 指向视频缓存的指针
           int mem_pitch,     // 每行多少个字节
           int alpha);      // alpha 等级
```

同样，与非 alpha 版本相比，惟一的不同是增加了参数 alpha，其取值必须是 0 到 NUN_ALPHA_LEVELS - 1。

（3）在仿射纹理映射函数中加入 alpha 混合功能

这与前面介绍的相同，但有三个这样的函数，它们的原型如下：

```
void Draw_Textured_Triangle_Alpha16(POLYF4DV2_PTR face,//指向多边形面的指针
           UCHAR *_dest_buffer, // 指向视频缓存的指针
           int mem_pitch,     // 每行多少个字节（320、640 等）
           int alpha);      // alpha 等级

void Draw_Textured_TriangleFS_Alpha16(POLYF4DV2_PTR face, //指向多边形面的指针
           UCHAR *_dest_buffer, // 指向视频缓存的指针
           int mem_pitch,     // 每行多少个字节（320、640 等）
           int alpha);      // alpha 等级

void Draw_Textured_TriangleGS_Alpha16(POLYF4DV2_PTR face, // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少个字节（320、640 等）
          int alpha);      // alpha 等级
```

现在有支持固定着色、恒定着色和 Gouraud 着色的纹理映射函数，它们都支持 alpha 混合（新增的参数为粗体）。作为一个例子，来看一下加入 alpha 混合功能之前和之后，支持固定着色的纹理映射函数的内循环：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  // 绘制像素
```

```
  screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
                ((vi >> FIXP16_SHIFT) << texture_shift2)];

  // 计算下一个像素的 u、v
  ui+=du;
  vi+=dv;
  } // end for xi
```

下面是加入 alpha 混合功能后的内循环代码：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  // 绘制像素
  screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] +
           alpha_table_src2[textmap[(ui >> FIXP16_SHIFT) +
                ((vi >> FIXP16_SHIFT) << texture_shift2)]];

  // 计算下一个像素的 u、v
  ui+=du;
  vi+=dv;
  } // end for xi
```

同样，只需将纹素和屏幕像素的颜色作为 alpha 查找表索引，然后将表查找结果相加，并将其绘制到屏幕上即可。下面是调用支持 alpha 混合的光栅化函数的渲染函数：

```
void Draw_RENDERLIST4DV2_Solid_Alpha16(
       RENDERLIST4DV2_PTR rend_list, // 渲染列表
       UCHAR *video_buffer,  // 指向视频缓存的指针
       int lpitch,       // 视频缓存中每行的字节数
       int alpha_override);  // 重新设置场景的 alpha 值
```

这个函数与以前的版本相同：使用渲染列表、指向视频缓存的指针和视频缓存的跨距来调用它，它将绘制渲染列表中所有的多边形。不过，它新增了一个参数：alpha_override。该参数用于设置绘制每个多边形时使用的 alpha 等级。如果 alpha_override 为-1，则不使用它；如果为 0 到 NUM_ALPHA_LEVELS – 1，将根据它来渲染透明多边形。

警告：光栅化函数接受一个 alpha 等级作为参数，但这个参数是如何得到的呢？现在是在调用渲染函数时指定的，稍后我们将把它存储到多边形的初始颜色中。

2. 在使用 z 缓存的光栅化函数中加入 alpha 混合功能

这与在不使用 z 缓存的光栅化函数中加入 alpha 混合功能的方法相同。除了由于 z 缓存测试导致逻辑方面不同外，没有其他的不同。使用 z 缓存时，如果前面透明的多边形先被绘制，位于它后面的多边形将不能通过 z 缓存测试，因此不被绘制，导致渲染结果不正确。为避免出现这种问题，可预先使用 z 排序将多边形按从后到前的顺序排列；也可以在渲染多边形的函数中加入额外的逻辑，以便总是先渲染被透明多边形遮住的多边形。将在本章后面出现这种问题时再去解决。现在，alpha 混合代码只管混合。

在所有使用 z 缓存的光栅化函数中加入 alpha 混合功能的方法都相同，因此这里只列出函数原型，然后通过一个例子来说明函数被修改前后的差别。下面是这些函数的原型：

```
void Draw_Textured_TriangleZB_Alpha16(POLYF4DV2_PTR face, // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少字节（320、640 等）
          UCHAR *zbuffer,    // 指向 z 缓存的指针
          int zpitch,       // z 缓存中每行多少字节
          int alpha);       // alpha 等级

void Draw_Textured_TriangleFSZB_Alpha16(POLYF4DV2_PTR face,// 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少字节（320、640 等）
          UCHAR *zbuffer,    // 指向 z 缓存的指针
          int zpitch,       // z 缓存中每行多少字节
          int alpha);       // alpha 等级
```

```
void Draw_Textured_TriangleGSZB_Alpha16(POLYF4DV2_PTR face,// 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,    // 每行多少字节（320、640 等）
          UCHAR *_zbuffer,   // 指向 z 缓存的指针
          int zpitch,      // z 缓存中每行多少字节
          int alpha);      // alpha 等级

void Draw_Triangle_2DZB_Alpha16(POLYF4DV2_PTR face, // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,      // 每行多少字节（320、640 等）
          UCHAR *zbuffer,     // 指向 z 缓存的指针
          int zpitch,       // z 缓存中每行多少字节
          int alpha);       // alpha 等级

void Draw_Gouraud_TriangleZB_Alpha16(POLYF4DV2_PTR face,  // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少字节（320、640 等）
          UCHAR *zbuffer,    // 指向 z 缓存的指针
          int zpitch,      // z 缓存中每行多少字节
          int alpha);      // alpha 等级
```

作为一个例子，来看一下被修改前后，最复杂的、支持 z 缓存和 alpha 混合的函数的内循环代码。这就是支持 Gouraud 着色和纹理映射的光栅化函数，它名为 Draw_Textured_TriangleGSZB_Alpha16()。加入 alpha 混合功能之前，该函数名为 Draw_Textured_TriangleGSZB_16()，其内循环代码如下：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  // 绘制像素
  // 检查像素的 z 值是否比 z 缓存中相应的 z 值小
  if (zi < z_ptr[xi])
    {
  // 首先读取纹素
   textel = textmap[(si >> FIXP16_SHIFT) +
             ((ti >> FIXP16_SHIFT) << texture_shift2)];

   // 提取 R、G、B 分量
   r_textel = ((textel >> 11)    );
   g_textel = ((textel >> 5) & 0x3f);
   b_textel =  (textel    & 0x1f);

   // 使用 Gouraud 着色处理得到的颜色对纹素进行调制
   r_textel*=ui;
   g_textel*=vi;
   b_textel*=wi;

   // 绘制像素
   screen_ptr[xi] = ((b_textel >> (FIXP16_SHIFT+8)) +
            ((g_textel >> (FIXP16_SHIFT+8)) << 5) +
            ((r_textel >> (FIXP16_SHIFT+8)) << 11));
   // 更新 z 缓存
   z_ptr[xi] = zi;
   } // end if

   // 计算下一个像素的 u、v、w、z、s 和 t
   ui+=du;
   vi+=dv;
   wi+=dw;

   zi+=dz;

   si+=ds;
   ti+=dt;

   } // end for xi
```

下面是加入 alpha 混合功能后的内循环代码：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  // 绘制像素
  // 检查像素的 z 值是否比 z 缓存中相应的 z 值小
  if (zi < z_ptr[xi])
    {
  // 首先读取纹素
   textel = textmap[(si >> FIXP16_SHIFT) +
              ((ti >> FIXP16_SHIFT) << texture_shift2)];

    // 提取 R、G、B 分量
    r_textel = ((textel >> 11)    );
    g_textel = ((textel >> 5) & 0x3f);
    b_textel =  (textel    & 0x1f);

    // 使用 Gouraud 着色得到的颜色对纹素进行调制
    r_textel*=ui;
    g_textel*=vi;
    b_textel*=wi;

    // 最后绘制像素
    screen_ptr[xi] = alpha_table_src1[screen_ptr[xi]] +
       alpha_table_src2[((b_textel >> (FIXP16_SHIFT+8)) +
              ((g_textel >> (FIXP16_SHIFT+8)) << 5) +
              ((r_textel >> (FIXP16_SHIFT+8)) << 11))];
    // 更新 z 缓存
    z_ptr[xi] = zi;

    } // end if

   // 计算下一个像素的 u、v、w、z、s 和 t
      ui+=du;
      vi+=dv;
      wi+=dw;

      zi+=dz;

      si+=ds;
      ti+=dt;

    } // end for xi
```

最后，介绍调用支持 alpha 混合和 z 缓存的光栅化函数的渲染函数，它名为 Draw_RENDERLIST4DV2_SolidZB_Alpha16()，原型如下：

```
void Draw_RENDERLIST4DV2_SolidZB_Alpha16(RENDERLIST4DV2_PTR rend_list,
      UCHAR *video_buffer,
      int lpitch,
      UCHAR *zbuffer,
      int zpitch,
      int alpha_override);
```

和以前一样，这个函数接受渲染列表、视频缓存及其跨距、z 缓存及其跨距作为参数，另外还新增了一个参数：alpha_override，它用于指定渲染透明多边形时使用的 alpha 等级，必须为 0 到（NUM_ALPHA_LEVELS - 1）或-1（不指定 alpha 等级）。

DEMOII12_2.CPP|EXE 是一个 alpha 混合演示程序，图 12.10 是其屏幕截图。

这个演示程序显示大量移动的物体，然后使用变量 alpha_override 在渲染列表级缓慢地提高和降低 alpha 混合因子（我们还不能在物体级设置 alpha 等级）。您还可以启用/禁用 z 排序和其他选项。要编译该演示程序，需要包含所有的库模块（T3DLIB1-10.CPP|H）和 DirectX .LIB 文件，并生成 Win32 .EXE 目标文件。



图 12.10　alpha 混合演示程序的屏幕截图

12.4.2　在物体级支持 alpha 混合功能

接下来需要解决的一个问题是，提供一种设置多边形的 alpha 分量（透明度）的方法。我们在设计颜色数据结构时就考虑到了这一点，该数据结构是在 T3DLIB6.CPP|H 中定义的：

```
typedef struct RGBAV1_TYP
{
union
  {
  int rgba;           // 压缩格式
  UCHAR rgba_M[4];        // 数组格式
  struct { UCHAR a,b,g,r; }; // 显式名称格式
  }; // end union

} RGBAV1, *RGBAV1_PTR;
```

上述数据结构用于表示 32 位的颜色，它支持 8 位的 alpha 值和 RGB 8.8.8 格式。不幸的是，在多边形数据结构中，并没有使用上述数据类型，而是使用一个 32 位的整数类型来存储 16 位格式的颜色，根本没有涉及 alpha。尽管如此，还是有办法的。当前，RGB 颜色是 16 位的值，它存储在多边形数据结构中 color 字段的后 16 位中，因此我们随意使用 color 字段的前 16 位。我们将把 alpha 分量存储在 color 字段的前 8 位中；另外，使用下述属性来指出多边形有 alpha 分量：

```
#define POLY4DV2_ATTR_TRANSPARENT      0x0002
```

该属性是在 T3DLIB7.CPP|H 中定义的。要在多边形中支持 alpha 混合，只需将多边形的属性设置为 POLY4DV2_ATTR_TRANSPARENT 和其他所需的属性，并将 alpha 等级存储在 color 字段的前 8 位中，如图 12.11 所示。

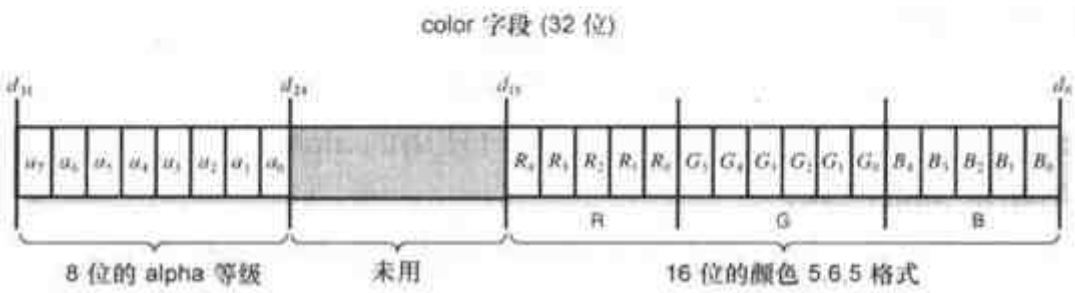


图 12.11　将 alpha 等级存储在 color 字段的前 8 位中

什么事情都不可能一帆风顺，我们现在面临的难题是，需要重新编写物体加载函数，这是无法避免的。除此之外，还需要拿出一种将多边形标记为透明的方法。幸运的是，Caligari trueSpace .COB 文件格式中有一个透明类型（transparency class），使用它即可。和以往一样，需要指定一种规则，以便以半自然的方式

使用透明类型来传递所需的信息，作者可不想使用粉红色来表示透明！

首先，需要确定各种版本的 Caligari 建模程序对透明性的支持情况。在 4.0 以上的版本中，通过界面提供了我们所需的支持，因此万事大吉。现在复习一下我们之前是如何对着色模型进行编码的，规则如下：

1．所有的多边形都必须是三角形。

2．对于没有纹理的多边形，其颜色类型（color class）必须设置为 plain color。

3．对于带纹理的多边形，其颜色类型必须设置为 texture map。另外，每个物体只能有一个纹理，纹理的重复因子必须是(1.0，1.0)。

4．多边形的着色模式是在反射类型（reflectance class）中设置的。constant 表示固定着色；matte 表示恒定着色处理；plastic 表示 Gouraud 着色。

这些约定很不错，我们可以非常自然地进行建模，指定所需的信息。要支持透明性，我们将使用 Caligari .COB 格式中的透明类型。图 12.12 是在 Caligari trueSpace 中设置透明性时的屏幕截图。

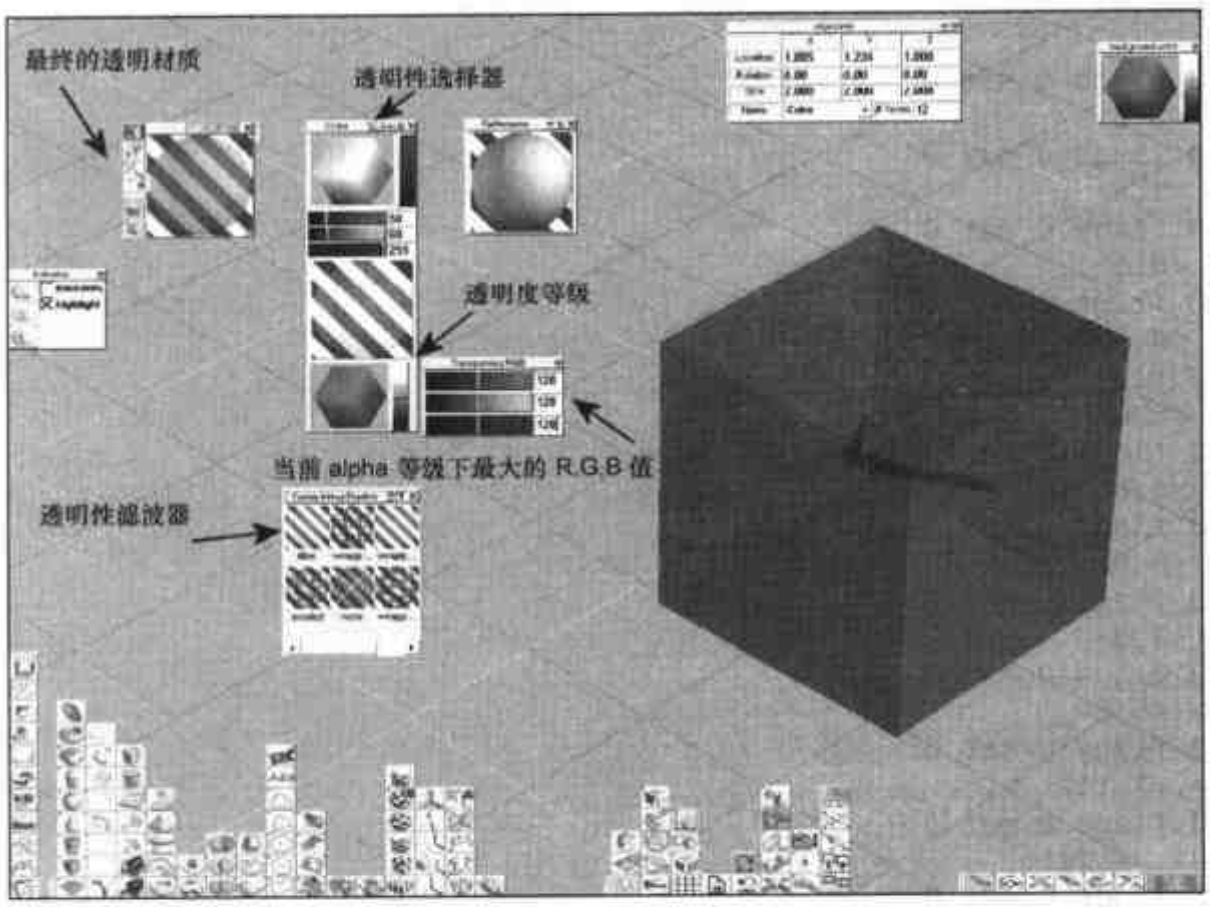


图 12.12　在 Caligari trueSpace 中设置模型的 alpha 值

我们将透明类型设置为 filter，然后将打开透明性控件，使用滑动块来控制多边形（面）的透明度（alpha 值）。例如，图 12.12 中的蓝色立方体的颜色类型被设置为 plain color（没有纹理），反射类型为 matte（恒定着色），透明类型为 filter，透明度被设置为 50%。下面是系统输出的立方体文件（作者将其命名为 CUBEBLUEALPHA_01.COB，并删除了一些无关的数据）：

```
Caligari V00.01ALH
Name Cube
center 1.08471 1.23355 1
x axis 1 0 0
y axis 0 1 0
z axis 0 0 1
Transform
1 0 0 1.08471
0 1 0 1.23355
0 0 1 0
0 0 0 1
World Vertices 8
-1.000000 -1.000000 0.000000
-1.000000 -1.000000 2.000000
1.000000 -1.000000 0.000000
1.000000 -1.000000 2.000000
-1.000000 1.000000 0.000000
```

```
1.000000 1.000000 0.000000
1.000000 1.000000 2.000000
-1.000000 1.000000 2.000000
Texture Vertices 14
0.000000 0.333333
0.000000 0.666667
0.250000 0.333333
0.250000 0.666667
0.500000 0.000000
0.500000 0.333333
0.500000 0.666667
0.500000 1.000000
0.250000 0.000000
0.250000 1.000000
0.750000 0.333333
0.750000 0.666667
1.000000 0.333333
1.000000 0.666667
Faces 12
Face verts 3 flags 0 mat 0
<0,0> <1,1> <3,3>
Face verts 3 flags 0 mat 0
<0,0> <3,3> <2,2>
Face verts 3 flags 0 mat 0
<0,8> <2,2> <5,5>
Face verts 3 flags 0 mat 0
<0,8> <5,5> <4,4>
Face verts 3 flags 0 mat 0
<2,2> <3,3> <6,6>
Face verts 3 flags 0 mat 0
<2,2> <6,6> <5,5>
Face verts 3 flags 0 mat 0
<1,9> <7,7> <6,6>
Face verts 3 flags 0 mat 0
<1,9> <6,6> <3,3>
Face verts 3 flags 0 mat 0
<4,10> <5,5> <6,6>
Face verts 3 flags 0 mat 0
<4,10> <6,6> <7,11>
Face verts 3 flags 0 mat 0
<0,12> <4,10> <7,11>
Face verts 3 flags 0 mat 0
<0,12> <7,11> <1,13>
DrawFlags 0
Unit V0.01 Id 18620117 Parent 18620116 Size 00000009
Units 2
ObRQ V0.01 Id 18620121 Parent 18620116 Size 00000121
Object Radiosity Quality: 0
Object Radiosity Max Area: 0
Object Radiosity Min Area: 0
Object Radiosity Mesh Accuracy: 0
Mat1 V0.06 Id 18855732 Parent 18620116 Size 00000099
mat# 0
shader: phong facet: auto32
rgb 0.227451,0.235294,1
alpha 1 ka 0.1 ks 0.1 exp 0 ior 1
ShBx V0.03 Id 18855733 Parent 18855732 Size 00000462
Shader class: color
Shader name: "plain color" (plain)
Number of parameters: 1
colour: color (58, 60, 255)
Flags: 3
Shader class: transparency
Shader name: "filter" (plain)
Number of parameters: 1
colour: color (128, 128, 128)
Flags: 3
```

```
Shader class: reflectance
Shader name: "matte" (matte)
Number of parameters: 2
ambient factor: float 0.1
diffuse factor: float 1
Flags: 3
Shader class: displacement
Shader name: "none" (none)
Number of parameters: 0
Flags: 3
END V1.00 Id 0 Parent 0 Size    0
```

其中我们现在使用的材质类型被显示为粗体。我们只需在.COB 加载函数中加入读取另一个类型（透明类型）的功能，然后对 colour 字段进行分析，从中提取 alpha 等级。但在此之前，需要说明另一个细节。在大多数建模程序（包括 trueSpace）中，alpha/透明度都由三个分量(R、G、B)组成。我们还没有介绍基于颜色的 alpha，我们只是使用 alpha 分量。这里将 R、G、B 分量中最大的值作为 alpha 值。您可以使用 R、G、B 或它们的组合作为 alpha 值，但加载函数将 R、G、B 分量中最大的值作为 alpha 值。最后，.COB 文件中的 alpha 值的取值范围为 0～255，我们必须将其缩放到 0 到（NUM_ALPHA_LEVELS - 1）。现在，我们便在模型级提供了 alpha 混合和透明性支持。下面是新的.COB 加载函数的原型，该函数支持 alpha：

```
int Load_OBJECT4DV2_COB2(OBJECT4DV2_PTR obj, // 指向物体的指针
             char *filename,    // Caligari COB 文件的名称
             VECTOR4D_PTR scale, // 初始缩放因子
             VECTOR4D_PTR pos,   // 初始位置
             VECTOR4D_PTR rot,   // 初始旋转角度
             int vertex_flags,   // 顶点顺序反转和变换标记
             int mipmap=0);  // mipmapping 启用标记
                       //1 表示生成 mip 映射，0 表示不生成
```

除新增了参数 mipmap（将在后面介绍）外，该函数与前一个版本 Load_OBJECT4DV2_COB()相同。在函数内部，新增了读取.COB 模型的透明类型的代码。该函数的代码如下：

```
////////////////////////////////////////////////////////////////////////////
// ADDED CODE FOR TRANSPARENCY AND ALPHA BLENDING ////////////////////////
////////////////////////////////////////////////////////////////////////////
// 现在需要知道材质中是否包含透明信息
// 我们决定将使用这种透明类型来设置这种信息
// 为此，必须使用将透明性设置为"filter"，然后使用 RGB 颜色值来设置透明度
// 其中(0,0,0)表明完全透明，(255,255,255) 表示完全不透明

// 为读取透明信息，需要在文件中查找类似于下面的内容：
// Shader class: transparency
// Shader name: "filter" (plain)
// Number of parameters: 1
// colour: color (146, 146, 146)
//
// 如果不是透明的，将看到如下信息：
//
// Shader class: transparency
// Shader name: "none" (none)
// Number of parameters: 0
//
// 我们将 R、G、B 中最大的值用作 alphe/透明度值
// 因此如果 R、G、B 为(255, 255, 255)，则认为多边形完全不透明

//  查找"Shader class: transparency"
while(1)
   {
   // 读取下一行
   if (!parser.Getline(PARSER_STRIP_EMPTY_LINES | PARSER_STRIP_WS_ENDS))
    {
    Write_Error("\nshader transparency class not found in .COB file %s.",
          filename);
    return(0);
    } // end if
```

```
    // 查找"Shader class: transparency"
    if (parser.Pattern_Match(parser.buffer,
                "['Shader'] ['class:'] ['transparency']") )
     {
     // 至此，我们知道下一个"shader name"就是我们所需的
     // 因此，退出循环
     break;
     } // end if

    } // end while

while(1)
   {
   // 读取下一行
   if (!parser.Getline(PARSER_STRIP_EMPTY_LINES | PARSER_STRIP_WS_ENDS))
    {
    Write_Error("\nshader name ended abruptly! in .COB file %s.",
          filename);
    return(0);
    } // end if

   // 删除引号
   ReplaceChars(parser.buffer, parser.buffer, "\"",' ',1);

   // 找到了"name"吗?
   if (parser.Pattern_Match(parser.buffer, "['Shader'] ['name:'] [s>0]" ) )
    {
    // 确定透明性是否被启用
    if (strcmp(parser.pstrings[2], "none") == 0)
      {
      // 将 alpha 属性位设置为 0
      RESET_BIT(materials[material_index + num_materials].attr,
           MATV1_ATTR_TRANSPARENT);

      // 将 alpha 等级设置为 0
      materials[material_index + num_materials].color.a = 0;

      } // end if
   else
   if (strcmp(parser.pstrings[2], "filter") == 0)
    {
    // 将 alpha 属性位设置为 1
    SET_BIT(materials[material_index + num_materials].attr,
        MATV1_ATTR_TRANSPARENT);

    // 查找"color"行，从中提取 alpha 值
    while(1)
       {
       // 读取下一行
       if (!parser.Getline(PARSER_STRIP_EMPTY_LINES |
                 PARSER_STRIP_WS_ENDS))
        {
        Write_Error("\ntransparency color not found in .COB file %s.",
              filename);
        return(0);
        } // end if

       // 删除无关的字符
       ReplaceChars(parser.buffer, parser.buffer, ":(,)",' ',1);
       // 查找类似于"colour: color (146, 146, 146)"的内容
       if (parser.Pattern_Match(parser.buffer,
              "['colour'] ['color'] [i] [i] [i]") )
       {
        // 将 alpha 等级设置为 R、G、B 中最大的
        int max_alpha = MAX(parser.pints[0], parser.pints[1]);
        max_alpha = MAX(max_alpha, parser.pints[2]);
```

```
        // 设置 alpha 值
        materials[material_index + num_materials].color.a =
         (int)( (float)max_alpha/255 *
             (float)(NUM_ALPHA_LEVELS-1) + (float)0.5);

        // 截取
        if (materials[material_index + num_materials].color.a
           >= NUM_ALPHA_LEVELS)
       materials[material_index + num_materials].color.a =
             NUM_ALPHA_LEVELS-1;

        break;
        } // end if

       } // end while

      } // end if

     break;
    } // end if

  } // end while
```

其中对 alpha 等级进行转换的代码为粗体。在文件加载函数的最后，将多边形的属性设置为透明的，以便渲染函数调用支持 alpha 混合的光栅化函数来渲染它。下面是设置 alpha 分量和透明属性的代码——请注意 alpha 分量的存储位置，它存储在 color 字段的前 8 位中：

```
// now test for alpha channel
if (materials[ poly_material[curr_poly] ].attr & MATV1_ATTR_TRANSPARENT)
  {
  // set the value of the alpha channel,
  // upper 8-bits of color will be used to hold it
  // lets hope this doesn't break the lighting engine!
  obj->plist[curr_poly].color +=
     (materials[ poly_material[curr_poly] ].color.a << 24);
        // set the alpha flag in polygon
        SET_BIT(obj->plist[curr_poly].attr, POLY4DV2_ATTR_TRANSPARENT);
       } // end if
```

本章前面介绍了基于渲染列表的 alpha 混合，在这种情况下，我们不能通过物体来控制 alpha 等级，而只是设置 alpha_override 的值，然后调用渲染列表函数，将每个多边形都视为透明的，并使用 alpha_override 的值作为 alpha 等级。现在，我们能够在多边形级支持 alpha，演示程序 DEMOII12_3.CPP|EXE 是一个控制 alpha 和透明度的例子。它加载大量透明的球体以及一个以黑白相间棋盘作为纹理的实心球体，这些球体在一个 2D 图像前移动，图 12.13 是该演示程序的屏幕截图。和以往一样，这个演示程序也有帮助菜单。请读

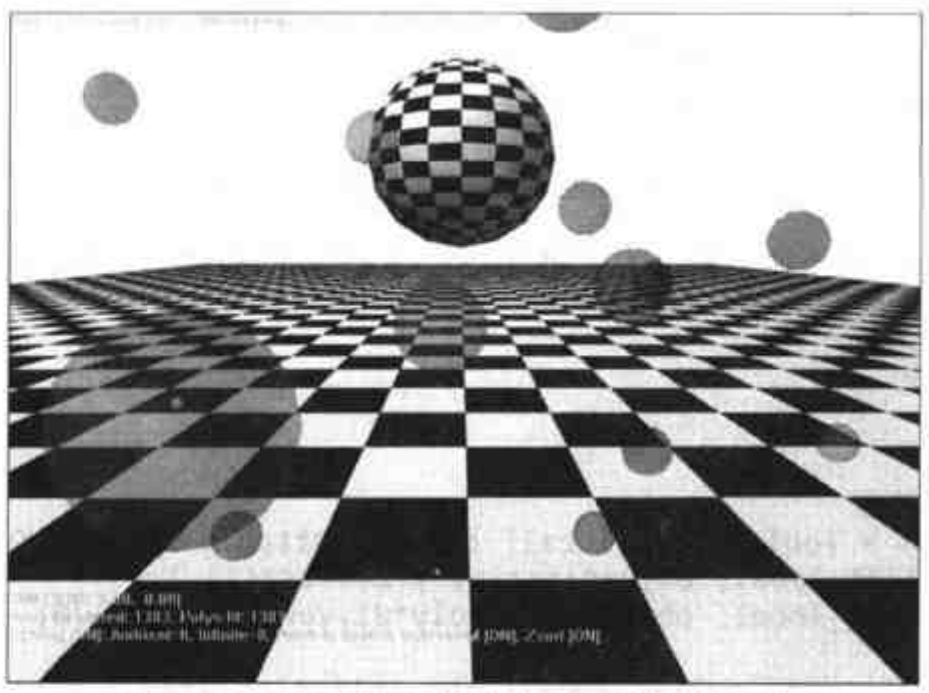


图 12.13　基于多边形/物体的 alpha 混合演示程序

者务必尝试分别使用 A 和 L 键关闭环境光源和无穷远光源。

12.4.3 在地形生成函数中加入 alpha 支持

现在我们能够在多边形级控制 alpha 值了，接下来将其用于地形生成函数中。作者想修改摩托艇模拟程序，使水是透明的。为此，可以采用某种方式在高程图中将多边形标记为透明的，并指定透明度。然而更简单的方法是，将“海平面”和 alpha 等级作为参数传递给地形生成函数；在该函数生成地形时，将位于海平面下的多边形设置为透明的，并将 alpha 值存储在多边形的 color 字段中。为修改地形生成函数，使之支持上述功能，最多只需 5 分钟。下面是 2.0 版地形生成函数的原型：

```
int Generate_Terrain2_OBJECT4DV2(OBJECT4DV2_PTR obj,   // 指向物体的指针
    float twidth,      // 地形宽度
    float theight,      // 地形长度
    float vscale,      // 最大可能高度
    char *height_map_file, // 256 色高程图的文件名
    char *texture_map_file,// 纹理图的文件名
    int rgbcolor,      // 没有纹理时地形的颜色
    VECTOR4D_PTR pos,    // 初始位置
    VECTOR4D_PTR rot,    // 初始旋转角度
    int poly_attr,     // 着色属性
    float sea_level,    // 海平面高度
    int alpha);         // 海平面下的多边形的 alpha 值
```

该函数新增了两个参数：

- float sea_level——海平面，高度小于这个值的多边形都将被设置为透明的。
- int alpha——取值为 0～255，表示水的透明度（0 表示完全透明，255 表示完全不透明）。在函数中，将把这个值缩放为 0 到（NUM_ALPHA_LEVELS – 1）。

要启用透明性支持，将 alpha 设置为大于-1 的整数。这样，函数将根据 sea_level 的值决定将哪些多边形设置为透明的，如图 12.14 所示。

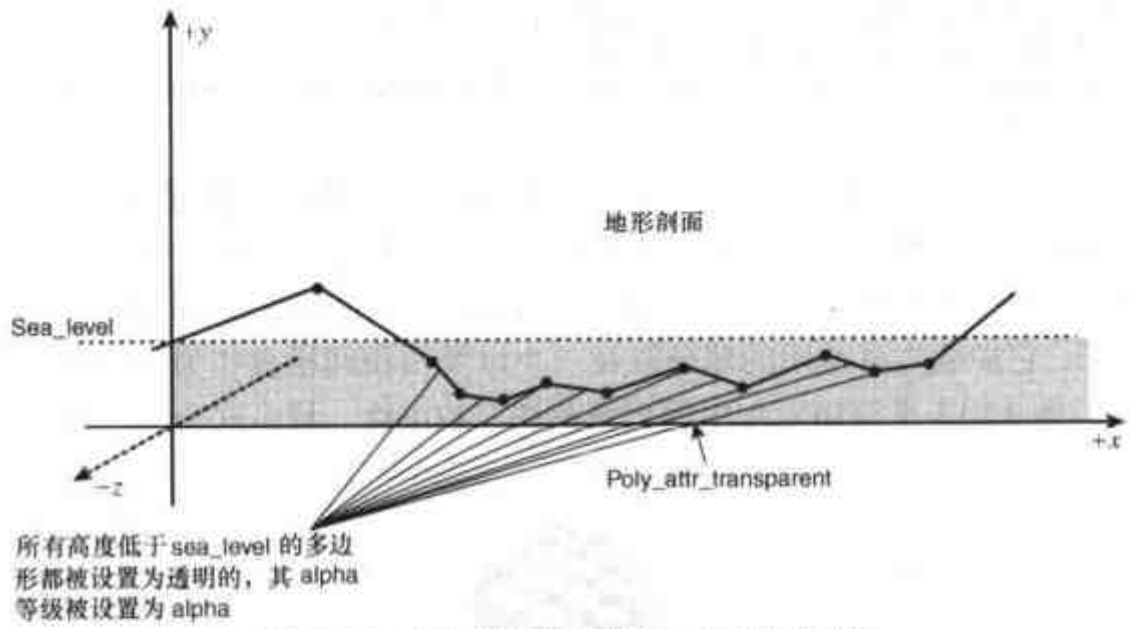


图 12.14 在地形生成函数中加入透明性支持

除了新增的功能外，地形生成函数没有其他变化。下面是支持 alpha 分量的代码，来自地形生成函数的顶点生成部分：

```
// 检查 alpha 的值
if (alpha >=0 )
  {
  // 计算两个多边形的高度
  float avg_y_poly1 = (obj->vlist_local[ obj->plist[poly*2].vert[0] ].y +
            obj->vlist_local[ obj->plist[poly*2].vert[1] ].y +
            obj->vlist_local[ obj->plist[poly*2].vert[2] ].y )/3;

  float avg_y_poly2 = (obj->vlist_local[ obj->plist[poly*2+1].vert[0] ].y +
            obj->vlist_local[ obj->plist[poly*2+1].vert[1] ].y +
```

```
            obj->vlist_local[ obj->plist[poly*2+1].vert[2] ].y )/3;

    // 检测多边形 1 是否在海平面下
    if (avg_y_poly1 <= sea_level)
     {
     int ialpha = (int)( (float)alpha/255 *
           (float)(NUM_ALPHA_LEVELS-1) + (float)0.5);

     // 设置 alpha 值
     obj->plist[poly*2+0].color+= (ialpha << 24);

     // 设置多边形的透明标记
     SET_BIT(obj->plist[poly*2+0].attr, POLY4DV2_ATTR_TRANSPARENT);
     } // end if

     // 检测多边形 2 是否在海平面下
     if (avg_y_poly2 <= sea_level)
      {
      int ialpha = (int)( (float)alpha/255 *
            (float)(NUM_ALPHA_LEVELS-1) + (float)0.5);

      // 设置 alpha 值
      obj->plist[poly*2+1].color+= (ialpha << 24);

      // 设置多边形的透明标记
      SET_BIT(obj->plist[poly*2+1].attr, POLY4DV2_ATTR_TRANSPARENT);
      } // end if

    } // end if
```

为创建演示程序，作者对前一章的摩托艇演示程序进行修改，调用 2.0 版的地形生成函数，并调用支持 z 缓存和 alpha 混合的渲染函数。

另外，还新增了一项特性，这进一步降低了演示程序的速度，但这项特性是必不可少的。水变成透明的后，天空背景是可见的，这是大忌，必须避免。因此，作者在地形后面放置了另一个地形，用作海底。这样，玩家透过海水看到的将是海底，而不是天空背景。实际上，我们渲染了两个地形。然而，这没有让游戏慢到无法玩的程度；但为使其速度足够快，作者将分辨率降低到了 640×480。可以使用 T 键在显示/不显示第二个地形之间进行切换，但如果不显示第二个地形，透过海水看到的将是云彩。图 12.15 是演示程序 DEMOII12_4.CPP|EXE 运行时的屏幕截图。

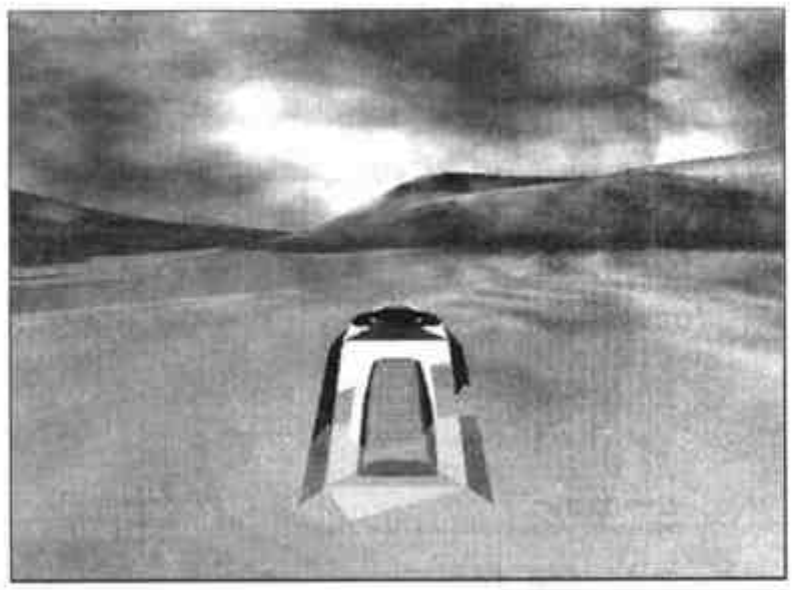


图 12.15　海水透明且包含海底地形的摩托艇演示程序

运行该演示程序时，透过海水可看到海底。该程序的控制方式与以前的摩托艇演示程序相同：

- 回车键——发动马达；
- 右箭头键——右转；
- 左箭头键——左转；
- 上箭头键——踩油门；
- T——在显示/不显示海底地形之间切换。

另外，大部分帮助菜单和引擎控制方式仍然适用：

- Z——开/关 z 缓存；
- S——开/关 z 排序；
- D——在常规渲染方式和 z 缓存过滤器之间切换；
- H——开/关帮助菜单；

● Esc——退出演示程序。

请尝试开/关各种光源、z 缓存、z 排序等。要编译该程序，除主程序 DEMOII12_4.CPP|H 外，还需要库模块 T3DLIB1-10.CPP|H 以及 DirectX .LIB 文件。

12.5 透视修正纹理映射和 1/z 缓存

我们终于为解决透视修正难题做好了准备。之所以现在才讨论透视修正纹理映射，是因为它依赖于很多基本概念，如 z 缓存和仿射纹理映射；必须先介绍它们。需要先介绍仿射纹理映射的原因是显而易见的，但需要先介绍 z 缓存的原因可能不那么明显。在前一章，我们通过分析证实，1/z 在屏幕空间中是呈线性变化的，并打算使用 1/z 缓存来代替 z 缓存，但没有实现它。实际上，这里介绍这些工具，旨在不仅让读者能够实现透视纹理映射，还能够真正地理解它。作者完全可以提供一个最优的 SIMD（单指令多数据，奔腾 III 以上处理器支持的新指令）透视修正纹理映射函数，但这样做对读者毫无帮助。

本节将介绍透视修正纹理映射的数学推导（使用两种不同的方式）、创建使用 1/z 缓存的光栅化函数，然后基于这些思想实现完美透视修正和分段线性透视修正纹理映射函数。最后，讨论其他一些近似技术和优化方法。

12.5.1 透视纹理映射的数学基础

仿射纹理映射、Gouraud 着色和三角形光栅化函数都是基于线性插值的，如图 12.16 所示。

这样的工作我们做过很多次。要沿三角形从上到下通过线性插值计算某个值，将三角形均匀地分割成多个部分，然后从上到下进行插值。一般而言，要通过插值计算 y0 和 y1 之间的 d 值，可以这样做：

```
float d = y0;
float dy = y1 - y0;

float dxdy = d/dy;

for (y=y0; y <= y1; y++)
  {
  // 使用 d

  // 计算下一个 d 值
  d+=dxdy;

  } // end for
```

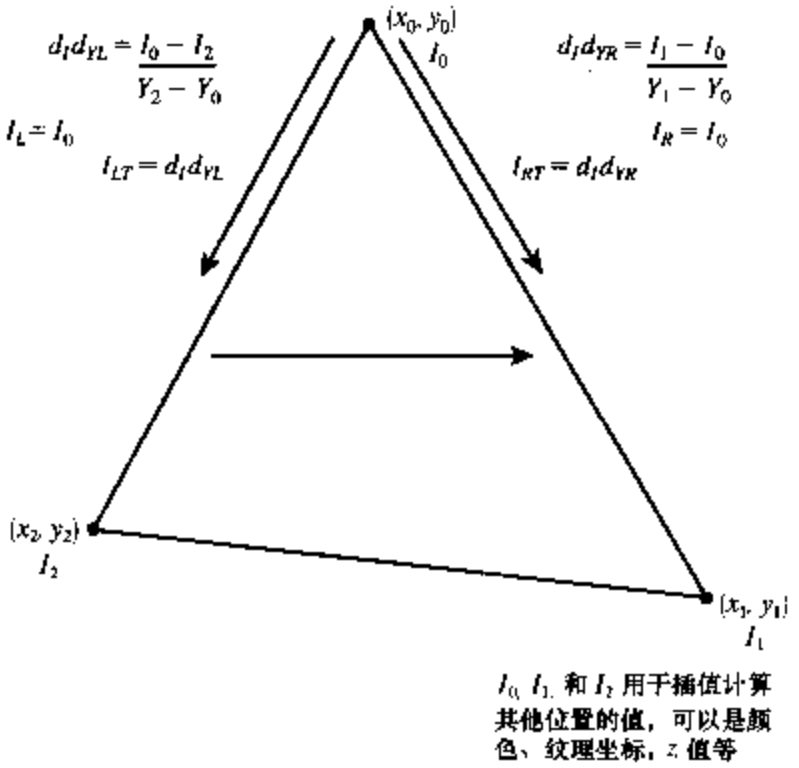


图 12.16 标准插值流程

线性插值的问题在于，不能用它来插值计算在屏幕空间中不呈线性变化的量。因此，它不适用于纹理坐标、z 坐标，甚至颜色强度。投影到屏幕时使用的是透视变换，这扭曲了上述数据；但扭曲程度都是可以接受的。从远处观看时，仿射纹理映射的效果很不错；使用线性插值时 Gouraud 着色的效果也不太糟糕；而 z 缓存在大多数时候也管用。然而，它们都是完全错误的。

前一章的分析表明，经过透视变换进入屏幕空间后，我们一直对其进行线性插值的那些量都不再是线性变化的，要让它们仍然是线性的，必须除以 z。也就是说，在屏幕空间中，只有 1/z 是线性变化的，这是所有问题的关键。下面介绍两种常见的透视修正纹理映射的推导方法，让读者能够真正理解其工作原理。

1. 透视修正公式推导方法 1

这种推导方法没有第二种方法严密，但更简明。如图 12.17 所示，我们要将一条线段投影到屏幕上。这条线段是 2D 的，但 3D 线段与此相同，只需假设位于 X-Z 平面内，即 $y=0$ 即可。这条线段的两个端点的坐标分别是 $(x1, z1)$ 和 $(x2, z2)$。可以用下列方程来表示这条线段：

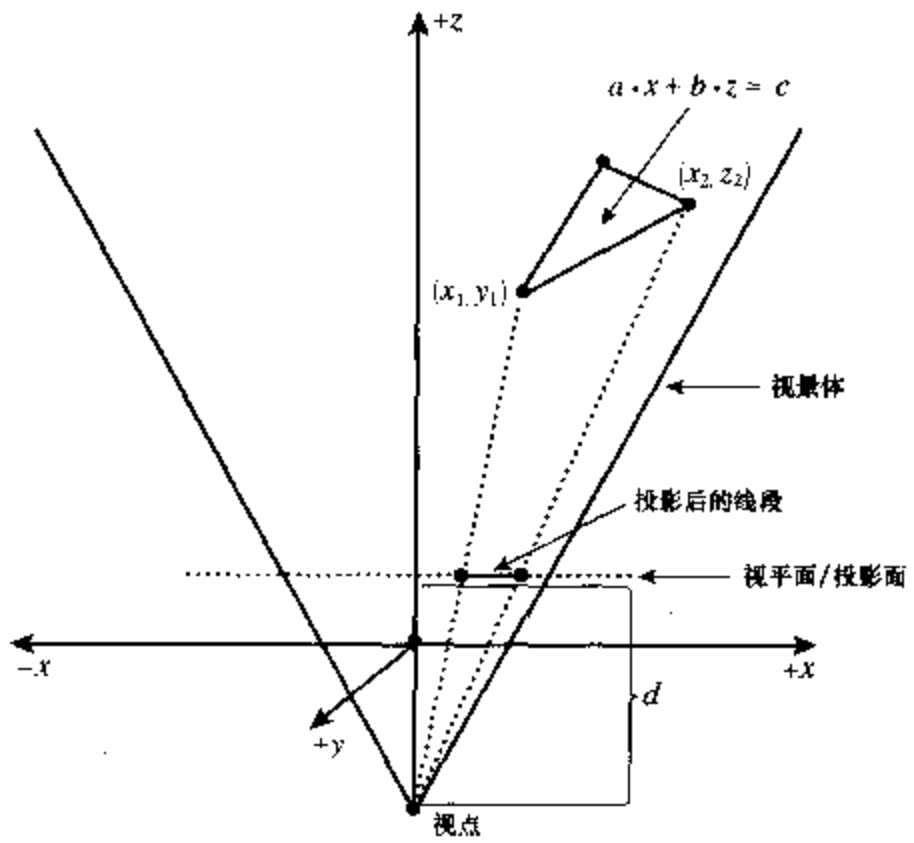


图 12.17　将 3D 线段投影到 2D 空间中

```
a*x + b*z = c
```

计算系数 a、b 和 c 的方法很多，例如可以将(x1，z1)和(x2，z2)代入上述方程以求解它们：

```
a*x1 + b*z1 = c
a*x2 + b*z2 = c
```

可以使用代入法、矩阵、高斯消元法或 TI-92 计算器来求解上述方程；但我们并不需要求解上述系数，而只是想使用这种形式的方程：

```
a*x + b*z = c
```

现在，假设要使用透视变换将线段上的点(x，z)投影到屏幕上，这意味着只需除以 z（当然还需要乘以视距，这里假设 d 为 1.0，以消除其影响）：

$x_{per} = x/z$

根据上述方程求解 x，结果为：

$x = z * x_{per}$

这相当于解答这样的问题：屏幕空间坐标 x_{per} 对应的 3D 空间是多少？接下来将 x 代入直线方程，结果如下：

$a*x + b*z = c \rightarrow a*(z*x_{per}) + b*z = c$

根据该方程求解 z，结果如下：

$z = c / (a*x_{per} + b)$

对等式两边求倒数，结果如下：

$1/z = (a/c)* x_{per} + (b/c)$

上述方程表明，$1/z$ 和 x_{per} 之间呈线性关系。将 $x_{per} = x/z$ 代入上述方程，结果如下：

$1/z = (a/c)* x/z + (b/c)$

这表明 $1/z$ 和 x/z 之间呈线性关系。这意味要计算 3D 空间中的 x、z、u 等，必须先将它们除以 z，再通过线性插值计算除以 z 后的值，然后将其除以 $1/z$，得到 3D 空间中的值。例如，让我们将纹理坐标 u 代入上述方程，看看如何进行线性插值：

$1/z = (a/c)* u/z + (b/c)$

我们首先计算 $1/z$，并令其为 z'；计算 u/z，并令其为 u'。然后执行线性插值，得到 u'和 z'，其中 u'表示

通过线性插值得到的 u/z，z'表示通过线性插值得到的 $1/z$。但我们要的是 u 而不是 u'，为此需要将 u'除以 z'：

```
u = u'/z' = (u/z) / (1/z) = (u/z) * (z/1) ≈ u
```

因此，我们只需通过插值计算 u'，并将其除以 z'来得到 u。当然，u 可以是任何量：纹理坐标、颜色值等。对于每个像素，根据 u'和 v'计算纹理坐标 u 和 v 需要执行两次除法运算，因此使用透视修正纹理映射时，每个像素增加了两次除法运算。

讨论另一种更严谨的推导方法之前，先从编程的角度来讨论这一点。显然，需要沿三角形边向下插值计算 $1/z$，因此可以从 z 缓存切换到 $1/z$ 缓存，而不丢失任何信息。使用 $1/z$ 缓存时，只需将内循环中的比较运算符小于改为大于即可。实际上，我们将重新考虑要使用的定点格式，以提高小数部分的精度。由于 $1/z$ 不可能大于 1.0，因此没有必要使用原来的 16.16 定点格式——用 16 位表示整数部分，用 16 位表示小数部分，而应将小数点位置左移，几乎将所有位都用来表示小数——格式 4.28 将是一种不错的选择，这将在实现 $1/z$ 缓存时讨论。妥善地处理好实现细节后，每个像素将只增加两次除法运算。这些运算是无法避免的，当然我们可以进行优化和近似。

下面通过一个范例来说明仿射纹理映射和透视修正的情况。这里只考虑在三角形的一条边上通过插值计算一个纹理坐标。两个端点 **p1**(100，100，20)和 **p2**(50，110，50)的纹理坐标 **u** 分别是 0 和 63，如图 12.18 所示。

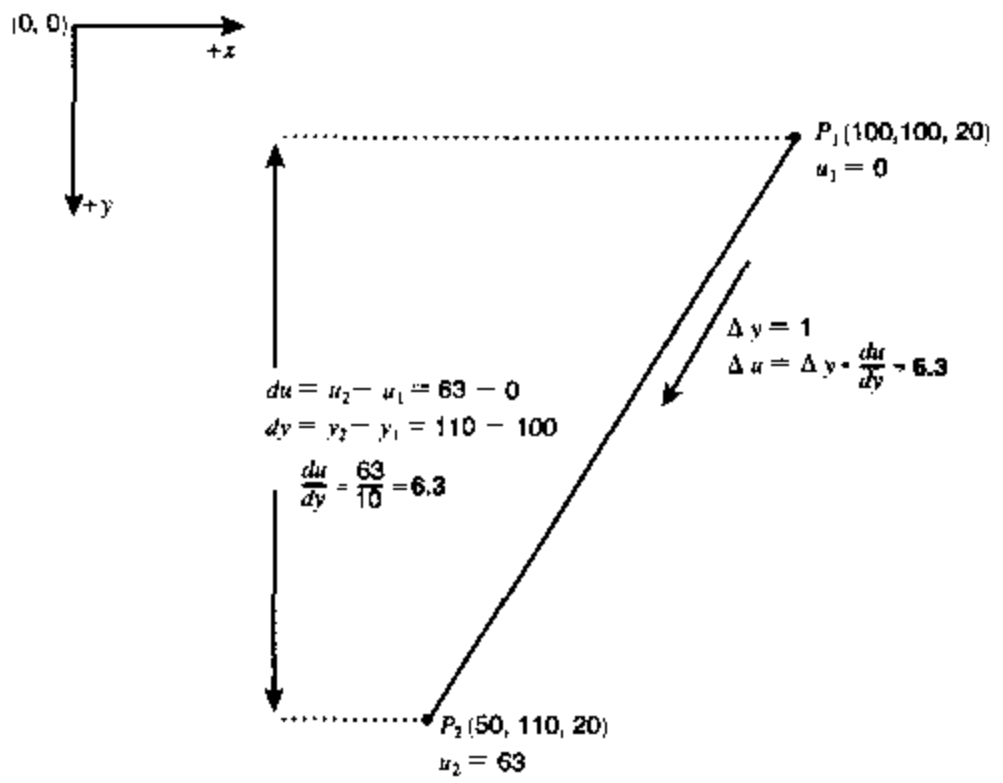


图 12.18　插值计算纹理坐标

在仿射纹理映射中，通过线性插值计算纹理坐标的算法如下：

```
float x1=100, y1=100, z1=20, u1=0,
   x2=50, y2=110, z2=50, u2=63;

float du = u2 - u1;
float dy = y2 - y1;
float dudy = du/dy;
// 计算初始值
float u = u1;

// 插值
for (int y=(int)y1; y <= (int)y2; y++)
  {
  printf("\n@y=%d, u=%f",y,u);

  // 计算下一个 u 值
  u+=dudy;
  } // end for
```

如果读者运行上述小程序，将得到如表 12.1 所示的错误数据。

表 12.1　　以线性插值方式计算得到的纹理坐标

y 坐标	纹理坐标	y 坐标	纹理坐标
100	$u = 0.000$	106	$u = 37.799$
101	$u = 6.300$	107	$u = 44.099$
102	$u = 12.600$	108	$u = 50.399$
103	$u = 18.900$	109	$u = 56.699$
104	$u = 25.200$	110	$u = 62.999$
105	$u = 31.500$		

在循环中，每次迭代的纹理坐标增量都相同，为 6.3；这里没有考虑 z 坐标，图 12.19 是根据上述数据绘制的图形。

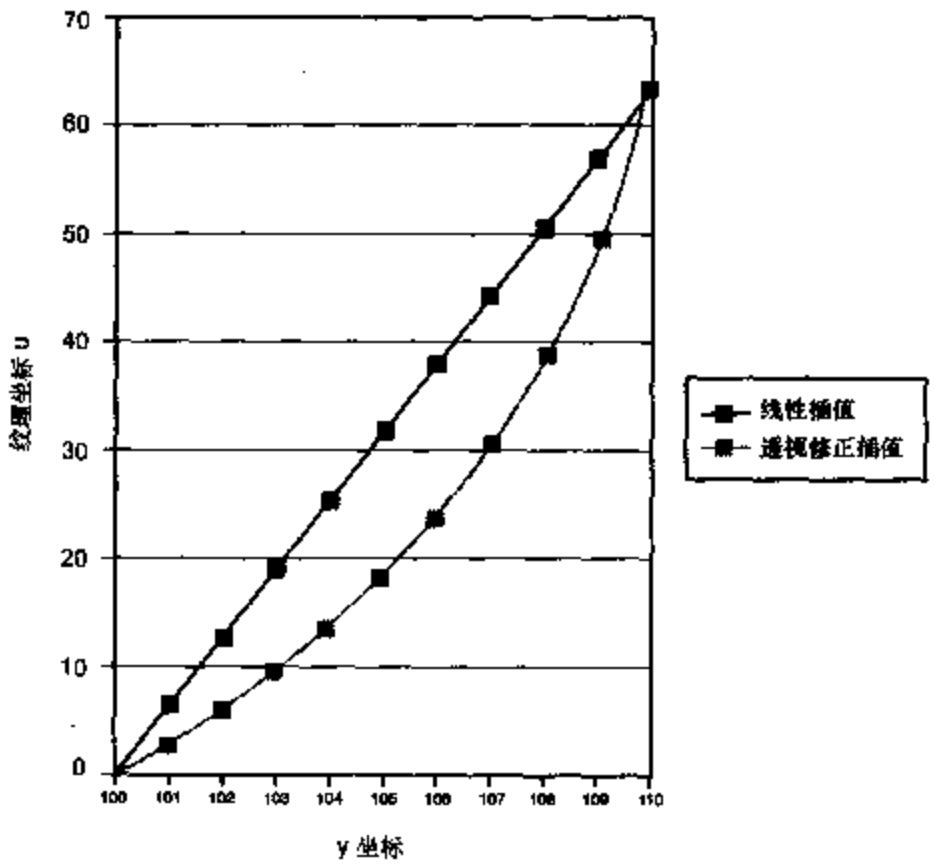


图 12.19　线性插值和透视修正插值得到的数据曲线

u 和 z 之间呈线性关系，这是错误的。现在来看一下该函数的透视修正版本，它在每次迭代中通过线性插值计算 $1/z$ 和 u/z，并将 u/z 除以 $1/z$ 来计算 u：

```
float x1=100, y1=100, z1=20, u1=0,
   x2=50, y2=110, z2=50, u2=63;

float du = u2/z2 - u1/z1;
float dy = y2 - y1;
float dz = 1/z2 - 1/z1;

float dzdy = dz/dy;
float dudy = du/dy;

// 计算初始值
float u = u1/z1;
float z = 1/z1;

// 插值
for (int y=(int)y1; y <= (int)y2; y++)
  {
  printf("\n@y=%d, 1/z=%f, u/z=%f: u=%f",y, u, z, u/z);

  // 计算下一个 u/z 值
  u+=dudy;

  // 计算下一个 1/z 值
  z+=dzdy;
  } // end for
```

从图 12.19 和表 12.2 可知，纹理坐标不再呈线性变化，而是随 1/z 而异。

表 12.2　　以透视修正插值方式计算纹理坐标

y 坐标	1/z	u/z	纹理坐标	y 坐标	1/z	u/z	纹理坐标
y = 100	1/z = 0.000	u/z = 0.050	u = 0.000	y = 106	1/z = 0.756	u/z = 0.032	u = 23.624
y = 101	1/z = 0.126	u/z = 0.047	u = 2.680	y = 107	1/z = 0.882	u/z = 0.029	u = 30.413
y = 102	1/z = 0.252	u/z = 0.044	u = 5.727	y = 108	1/z = 1.008	u/z = 0.026	u = 38.769
y = 103	1/z = 0.378	u/z = 0.041	u = 9.219	y = 109	1/z = 1.134	u/z = 0.023	u = 49.304
y = 104	1/z = 0.504	u/z = 0.038	u = 13.263	y = 110	1/z = 1.26	u/z = 0.020	u = 62.999
y = 105	1/z = 0.630	u/z = 0.035	u = 17.999				

为验证这种算法的正确性，假设两个端点的 z 坐标相同。在这种情况下，将没有透视扭曲，因为这两个端点之间的线段与投影面平行。因此，透视修正插值的结果应与仿射/线性插值的结果相同。来看看是不是这样。将 z1 和 z2 都设置为 20 以消除透视扭曲，然后运行透视修正插值程序，结果如表 12.3 所示。

表 12.3　　z 坐标相同时通过透视修正插值得到的纹理坐标

y 坐标	1/z	u/z	纹理坐标	y 坐标	1/z	u/z	纹理坐标
y = 100	1/z = 0.000	u/z = 0.050	u = 0.000	y = 106	1/z = 1.890	u/z = 0.050	u = 37.800
y = 101	1/z = 0.315	u/z = 0.050	u = 6.300	y = 107	1/z = 2.205	u/z = 0.050	u = 44.100
y = 102	1/z = 0.630	u/z = 0.050	u = 12.600	y = 108	1/z = 2.520	u/z = 0.050	u = 50.400
y = 103	1/z = 0.945	u/z = 0.050	u = 18.900	y = 109	1/z = 2.835	u/z = 0.050	u = 56.700
y = 104	1/z = 1.260	u/z = 0.050	u = 25.199	y = 110	1/z = 3.150	u/z = 0.050	u = 63.000
y = 105	1/z = 1.575	u/z = 0.050	u = 31.500				

如果对表 12.1 和 12.3 进行比较，将发现其中的纹理坐标相同（当然有很小的浮点误差），这表明我们的算法在任何情况下都是正确的。

2. 透视修正公式推导方法 2

这种推导方法与前一个方法类似，但更严密。这里将使用下述两点之间的插值公式：

```
p = (1 - t)*p1 + (t)*p2
```

注意：p 可以是向量，也可以是标量。

我们需要推导出用 z 表示的 p1 到 p2 之间的插值公式。图 12.20 说明了这种推导过程。

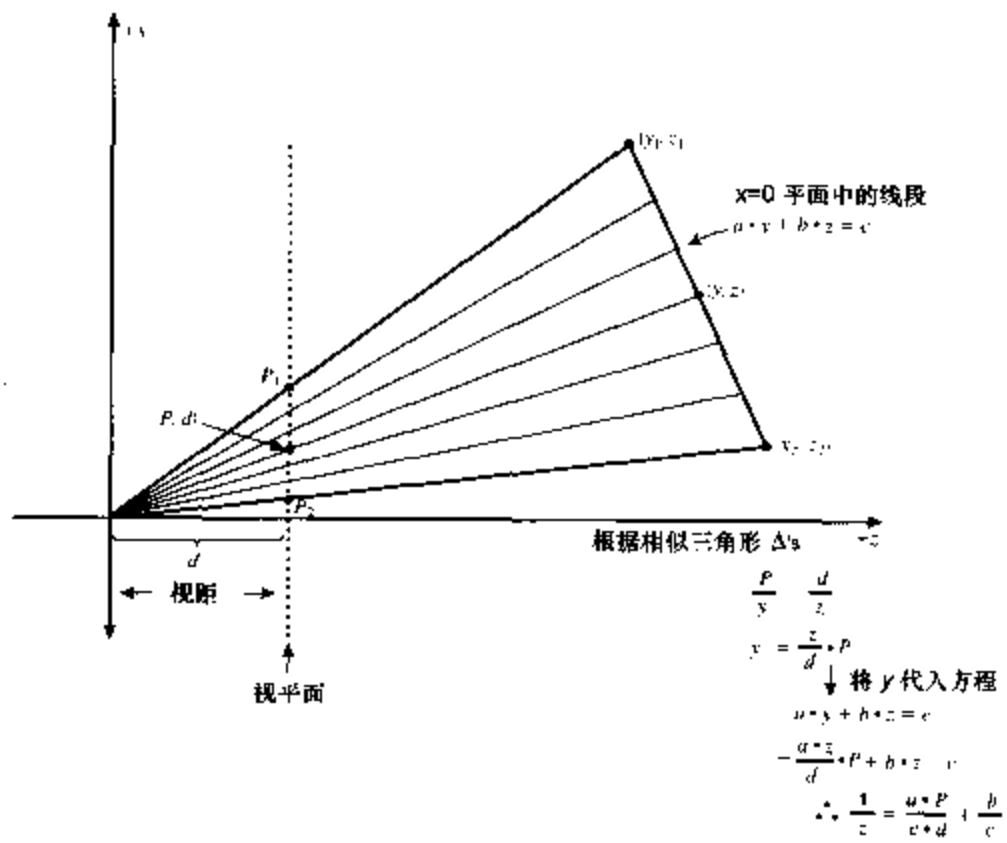


图 12.20　线段插值和透视投影

在图 12.20 中，有两个位于世界坐标空间中的点，它们的坐标分别是(*y*1，*z*1)和(*y*2，*z*2)。将这些点投影到 *z*=d 的视平面上时，得到点(p1，d)和(p2，d)。另外，线段上的任意一点(*y*，*z*)投影到视平面上时得到点(p，d)。

在 3D 空间中，位于 *y*-*z* 平面中(x=0)的直线的方程为：

```
a*y + b*z = c
```

请看图 12.20，根据三角形相似可知，交点(p，d)和 3D 点(*y*，*z*)具有如下关系：

```
p/y = d/z
```

求解 y，结果如下：

```
y = (p/d)*z
```

将其代入直线方程，结果如下：

```
(a*p/d)*z + b*z = c
```

将等式两边除以((a*p/d) + b)，结果如下：

```
z = c / ((a*p/d) + b)
```

在等式两边求倒数，结果如下：

```
1/z = ((a*p/d) + b)/c = (a*p/c*d) + (b/c)
```

将变量 p 与其他常量分离，结果如下：

```
1/z = (a/c*d)*p + (b/c)
```

因此，点(*y*1，*z*1)与其投影点(*p*1，*d*)之间存在如下关系：

```
1/z1 = (a/c*d)*p1 + (b/c)
```

点(*y*2，*z*2)与其投影点(*p*2，*d*)之间存在如下关系：

```
1/z2 = (a/c*d)*p2 + (b/c)
```

至此，我们确定了端点的情况，还需要确定中间点的情况。为此，将 p = (1 − *t*)*p1 + (*t*)*p2 代入下述方程：

```
1/z = (a/c*d)*p + (b/c)
```

结果如下：

```
1/z = (a/c*d)*[ (1-t)*p1 + (t)*p2 ] + (b/c)
```

计算乘法后结果如下：

```
1/z = (a/c*d)*p1*(1-t) + (a/c*d)*p2*(t) + (b/c)
```

接下来在等式的右边加上(b/c)*t*，再减去(b/c)*t*，得到如下结果：

```
1/z = [ (a/c*d)*p1 + (b/c) ]*(1-t) + [ (a/c*d)*p2 + (b/c) ]*(t)
```

请注意其中以粗体显示的项，它们分别与下述等式的右边相同：

```
1/z1 = (a/c*d)*p1 + (b/c)
1/z2 = (a/c*d)*p2 + (b/c)
```

因此，可以用这两个等式的左边（即 1/*z*1 和 1/*z*2）替换它们，最终结果如下：

```
1/z = [(1/z1)]*(1-t) + [(1/z2)]*(t)
```

这意味着可以在 1/*z*1 和 1/*z*2 之间进行线性插值。换句话说，任何 3D 空间量除以 z 的商在屏幕空间中都是呈线性变化的。

至此，读者应该知道为何 1/*z* 在屏幕空间中是呈线性变化的，同时任何 3D 空间量与 *z* 的商在屏幕空间中也是呈线性变化的。这一点应该很直观，因为进行屏幕投影时，我们将值除以 *z*。

12.5.2 在光栅化函数中加入 1/z 缓存功能

读者知道为支持透视修正纹理映射，需要在光栅化函数中通过线性插值计算 1/z 的原因后，接下来需要支持这种功能。这相当简单，但为提高精度，有些工作必须使用定点格式来完成。当前，我们使用的是 16.16 格式，这可以表示–32768 到 32768 的带符号整数，可以表示的最小小数为 $1/2^{16}$，这大约是 1.5258×10^{-5}，即可以精确到小数点后面 4～5 位。然而，这只是就格式本身而言的。执行乘除运算时，移位精度将急剧降低。因此，我们必须使用另一种定点格式，它更适合表示小于 1.0 的数，即 z 的倒数。现在的问题是，可以用多少位来表示小数部分。

答案是很多，但又不能太多。我们使用 32 位的定点数，其中 1 位用于表示正负号，这意味着还有 31 位可用。当然，如果要表示的数位于 0 到 1.99999999 之间，可以使用 2.30 格式，整数部分的 1 位用于表示正负号，余下的 1 位最大可表示 1，但整数部分的空间太紧张了。作者试验过这种格式，但经常会溢出，因此作者最终决定使用 4.28 格式，即 28 位用于表示小数部分，3 位用于表示整数部分，1 位用于表示正负号。这样，可以表示–8 到 7 之间的小数，小数部分的精度为 28 位（二进制），对于 1/z 缓存以及以后可能需要实现的插值来说，这足够了。为使用这种格式，在 T3DLIB10.H 中定义了下述变量：

```
#define FIXP28_SHIFT 28
```

作者以使用 z 缓存的光栅化函数为基础，将 z 缓存转换为 1/z 缓存。为编写支持透视修正的光栅化函数，以仿射纹理映射函数为基础，将线性纹理坐标插值改为 u/z 和 v/z 插值，并将插值结果除以 1/z 来得到像素的纹理坐标 u 和 v。稍后再介绍这些，先来看看最简单的 1/z 缓存光栅化函数，它绘制使用固定/恒定着色的多边形，且不支持 alpha 混合。该函数的原型如下：

```
void Draw_Triangle_2DINVZB_16(POLYF4DV2_PTR face, // 指向多边形面的指针
        UCHAR *_dest_buffer, // 指向视频缓存的指针
        int mem_pitch,     // 每行多少个字节(320、640 等)
        UCHAR *zbuffer,    // 指向 z 缓存的指针
        int zpitch);       // z 缓存中每行多少字节
```

该函数接受的参数与 z 缓存版本相同，但在内部使用 1/z 缓存。先来回顾一下 1/z 缓存的工作原理。如果有两个像素，其中一个像素的 z 值为 100，另一个为 200，则 z = 100 的像素应覆盖 z = 200 的像素。下面来计算这两个像素的 1/z。

对于 z = 100 的像素：

```
1/z = 1/100 = 0.01
```

对于 z = 200 的像素：

```
1/z = 1/200 = 0.005
```

因此，对于原来内循环中的比较代码：

```
if (zi < screen_ptr[x])
```

需要反转比较方向，使 1/z 更大时胜出：

```
if (zi > screen_ptr[x])
```

这在前一章介绍过了，读者不应对此感到惊讶。还需要修改在每一帧中都对 z 缓存进行初始化的代码。使用 z 缓存时，我们使用最大可能 z 值（如 32000）来填充 z 缓存，代码如下：

```
Clear_Zbuffer(&zbuffer, (32000 << FIXP16_SHIFT));
```

然而，我们不但使用了不同的定点格式，还需要使用 0 来填充 1/z 缓存。因此，需要将清空缓存的代码修改为：

```
Clear_Zbuffer(&zbuffer, 0);
```

提示：使用 1/z 缓存时，可以使用前面介绍过的优化方法：在每帧中给 1/z 加上一个偏移量，以免每帧都清除 1/z 缓存。然而，由于使用的是 4.28 格式，用于表示整数部分的只有 3 位，因此经过几帧后就将溢

出。作者决定现在暂时不使用这种优化方法。

接下来的一个问题是 z 缓存本身。需要对 z 缓存数据结构等进行修改吗？答案是否定的，z 缓存只不过是内存而已。然而，确实不能在使用 z 缓存和 1/z 缓存的函数之间共用缓存，否则将引起混乱。

接下来看一下范例光栅化函数中设置 1/z 缓存的代码。下面是使用 z 缓存时，设置顶点的代码：

```
// 设置顶点坐标
x0 = (int)(face->tvlist[v0].x+0.0);
y0 = (int)(face->tvlist[v0].y+0.0);

tz0 = (int)(face->tvlist[v0].z+0.5);

x1 = (int)(face->tvlist[v1].x+0.0);
y1 = (int)(face->tvlist[v1].y+0.0);

tz1 = (int)(face->tvlist[v1].z+0.5);

x2 = (int)(face->tvlist[v2].x+0.0);
y2 = (int)(face->tvlist[v2].y+0.0);

tz2 = (int)(face->tvlist[v2].z+0.5);
```

其中设置 z 坐标的代码为粗体。下面是使用 1/z 缓存时设置顶点的代码：

```
// 设置顶点坐标
x0 = (int)(face->tvlist[v0].x+0.0);
y0 = (int)(face->tvlist[v0].y+0.0);

tz0 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v0].z+0.5);

x1 = (int)(face->tvlist[v1].x+0.0);
y1 = (int)(face->tvlist[v1].y+0.0);

tz1 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v1].z+0.5);

x2 = (int)(face->tvlist[v2].x+0.0);
y2 = (int)(face->tvlist[v2].y+0.0);

tz2 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v2].z+0.5);
```

新增的内容有两项。首先，现在使用的是定点数。在上述代码中，将 1.0 转换为 4.28 定点格式，然后除以 z。因此，tz0、tz1 和 tz2 为定点数，而在支持 z 缓存的版本中不是。这样做的原因是，z 值为整数，如果直接用它来除以 1，将降低精度；因此执行除法运算之前，必须将 1 转换为定点数。现在，在插值计算 z 的代码中，不再需要左移 FIXP16_SHIFT 位了，因为这些值已经是定点格式了。这样，性能可能会稍有提高。由于 tz0、tz1、tz2 和 1/z 为 4.28 定点格式，因此在光栅化函数中无需进行转换。例如，以前通过插值计算 z 时，需要左移 FIXP16_SHIFT 位，以便将这些值转换为定点数：

```
dy = (y2 - y0);
dzdyl = ((tz2 - tz0) << FIXP16_SHIFT)/dy;
dzdyr = ((tz2 - tz1) << FIXP16_SHIFT)/dy;

dy = (min_clip_y - y0);

// 重新计算初始值
zl = dzdyl*dy + (tz0 << FIXP16_SHIFT);
zr = dzdyr*dy + (tz1 << FIXP16_SHIFT);
```

然而，使用 1/z 缓存时的代码如下：

```
dy = (y2 - y0);
dzdyl = ((tz2 - tz0))/dy;
dzdyr = ((tz2 - tz1))/dy;

dy = (min_clip_y - y0);

// 重新计算初始值
```

```
zl = dzdyl*dy + (tz0);
zr = dzdyr*dy + (tz1);
```

不再需要执行移位运算。在内循环中，除了反转了比较方向（以支持 1/z 缓存）外，代码与以前相同。这是因为无论是什么量，插值计算的代码都相同。读者需要牢记的是，设置顶点的方法不同：初始化为 1/z 而不是 z。下面是函数 Draw_Triangle_2DINVZB_16()中绘制所有扫描线的循环。这些代码表明，使用 1/z 缓存和使用 z 缓存的方法相同，如果有什么不同，那就是速度更快：

```
// 将指针 screen_ptr 指向起始行
screen_ptr = dest_buffer + (ystart * mem_pitch);

// 将指针 z-ptr 指向起始行
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi < yend; yi++)
  {
  // 计算扫描线端点的 x 坐标
  xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
  xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

  // 计算初始值
  zi = zl;

  // 计算增量
  if ((dx = (xend - xstart))>0)
      {
    dz = (zr - zl)/dx;
    } // end if
  else
    {
    dz = (zr - zl);
    } // end else

// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
   // 检查当前 z 值是否大于缓存中的 z 值
   if (zi > z_ptr[xi])
    {
    // 绘制像素
    screen_ptr[xi] = color;

    // 更新 z 缓存
    z_ptr[xi] = zi;
    } // end if
    // 计算下一个像素的 z 值
    zi+=dz;
    } // end for xi

// 计算下一条扫描线端点的 x、z 值
xl+=dxdyl;
zl+=dzdyl;

xr+=dxdyr;
zr+=dzdyr;

// 将指针 screen_ptr 指向下一行
screen_ptr+=mem_pitch;

 // 将指针 z_ptr 指向下一行
 z_ptr+=zpitch;

} // end for y
```

现在简要地复习一下，然后看一下所有使用 1/z 缓存的光栅化函数——支持透视修正的函数除外。为正确地实现透视修正纹理映射，需要使用 1/z 缓存；1/z 缓存不比 z 缓存逊色，它更可靠，同时又不会降低

性能。然而，为确保 1/z 的精度足够高，需要修改定点格式。对于离视平面很近的多边形，使用 1/z 能够更好地将它们区分开来，这样在可见性判定的准确度将更高。由于 1/z 值一开始就被转换为定点数，因此在光栅化函数中，不用像使用 z 缓存那样执行移位运算。最后，需要在每帧中使用 0 来填充 1/z 缓存，而不是像 z 缓存那样使用最大可能值来填充。这让我们可以使用偏移量技术，避免在每帧中都清空缓存，从而提高性能。

注意：在支持 alpha 混合的光栅化函数中加入 1/z 缓存功能的方法完全相同，这里不介绍其代码，但将列出它们的原型。

下面是支持 1/z 缓存的光栅化函数的原型。其中不包括支持透视修正的函数，这些函数稍后再介绍。

```
void Draw_Triangle_2DINVZB_16(POLYF4DV2_PTR face, // 指向多边形面的指针
        UCHAR *_dest_buffer, // 指向视频缓存的指针
        int mem_pitch,    // 每行多少字节（320、640 等）
        UCHAR *zbuffer,    // 指向 z 缓存的指针
        int zpitch);      // z 缓存中每行多少字节
```

该函数使用 1/z 缓存绘制标准的彩色三角形。

```
void Draw_Gouraud_TriangleINVZB_16(POLYF4DV2_PTR face,// 指向多边形面的指针
        UCHAR *_dest_buffer, // 指向视频缓存的指针
        int mem_pitch,     // 每行多少字节（320、640 等）
        UCHAR *zbuffer,    // 指向 z 缓存的指针
        int zpitch);      // z 缓存中每行多少字节
```

该函数使用 1/z 缓存绘制使用 Gouraud 着色的三角形。

```
void Draw_Textured_TriangleINVZB_16(POLYF4DV2_PTR face,//指向多边形面的指针
         UCHAR *_dest_buffer, // 指向视频缓存的指针
         int mem_pitch,     // 每行多少字节（320、640 等）
         UCHAR *zbuffer,     // 指向 z 缓存的指针
         int zpitch);      // z 缓存中每行多少字节
```

该函数使用 1/z 缓存和固定着色绘制使用仿射纹理映射的三角形。

```
void Draw_Textured_TriangleFSINVZB_16(POLYF4DV2_PTR face,// 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少字节（320、640 等）
          UCHAR *zbuffer,     // 指向 z 缓存的指针
          int zpitch);      // z 缓存中每行多少字节
```

该函数使用 1/z 缓存和恒定着色绘制使用仿射纹理映射的三角形。

```
void Draw_Textured_TriangleGSINVZB_16(POLYF4DV2_PTR face, // 指向多边形面的指针
           UCHAR *_dest_buffer,// 指向视频缓存的指针
           int mem_pitch, // 每行多少字节（320、640 等）
           UCHAR *_zbuffer,    // 指向 z 缓存的指针
           int zpitch);      // z 缓存中每行多少字节
```

该函数使用 1/z 缓存和 Gouraud 着色绘制使用仿射纹理映射的三角形。

```
void Draw_RENDERLIST4DV2_SolidINVZB_16(RENDERLIST4DV2_PTR rend_list,
           UCHAR *video_buffer, // 指向视频缓存的指针
           int lpitch,     // 每行多少字节（320、640 等）
           UCHAR *zbuffer, // 指向 z 缓存的指针
           int zpitch);   // z 缓存中每行多少字节
```

这是一个渲染列表函数，它根据多边形的着色和纹理映射信息调用相应的、使用 1/z 缓存的光栅化函数。

接下来列出支持 alpha 混合和 1/z 缓存的函数。除了参数列表末尾增加了参数 alpha_override 外，它们与非 alpha 混合版本相同。同样，这里没有列出支持透视修正的函数，因为我们还未介绍过。

```
void Draw_Triangle_2DINVZB_Alpha16(POLYF4DV2_PTR face, // 指向多边形面的指针
           UCHAR *_dest_buffer, // 指向视频缓存的指针
           int mem_pitch,     // 每行多少字节（320、640 等）
           UCHAR *zbuffer,    // 指向 z 缓存的指针
           int zpitch,       // z 缓存中每行多少字节
```

```
            int alpha);      // alpha 等级
```

该函数使用 1/z 缓存绘制标准的彩色三角形。

```
void Draw_Gouraud_TriangleINVZB_Alpha16(POLYF4DV2_PTR face, //指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少字节
            int alpha);       // alpha 等级
```

该函数使用 1/z 缓存绘制使用 Gouraud 着色的三角形。

```
void Draw_Textured_TriangleINVZB_Alpha16(POLYF4DV2_PTR face,//指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少字节
            int alpha);       // alpha 等级
```

该函数使用 1/z 缓存和固定着色绘制使用仿射纹理映射的三角形。

```
void Draw_Textured_TriangleFSINVZB_Alpha16(POLYF4DV2_PTR face,// 指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少字节（320、640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少字节
            int alpha);       // alpha 等级
```

该函数使用 1/z 缓存和恒定着色绘制使用仿射纹理映射的三角形。

```
void Draw_Textured_TriangleGSINVZB_Alpha16(POLYF4DV2_PTR face,  // 指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少字节（320、640 等）
            UCHAR *_zbuffer,   // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少字节
            int alpha);       // alpha 等级
```

该函数使用 1/z 缓存和 Gouraud 着色绘制使用仿射纹理映射的三角形。

```
void Draw_RENDERLIST4DV2_SolidINVZB_Alpha16(RENDERLIST4DV2_PTR rend_list,
            UCHAR *video_buffer,  // 指向视频缓存的指针
            int lpitch,       // 每行多少字节（320=640 等）
            UCHAR *zbuffer,    // 指向 z 缓存的指针
            int zpitch,       // z 缓存中每行多少字节
            int alpha_override); // alpha 溢出值
```

这是一个渲染列表函数，它根据多边形的着色和纹理映射信息调用相应的、使用 1/z 缓存的光栅化函数。DEMOII12_5.CPP|EXE 是一个 1/z 缓存技术的演示程序，图 12.21 是其屏幕截图。



图 12.21　1/z 缓存技术演示程序的屏幕截图

通过按 Z 键，可以在 1/z 缓存、z 缓存和不使用深度缓存之间进行切换；按 S 键可以开/关 z 排序。请尝试使用不同的缓存技术，然后将探测器移到物体中，看看哪种技术的精度更高。要编译该演示程序，需要 DEMOII12_5.CPP|H、T3DLIB1-10.CPP|H 和 DirectX .LIB 文件。

提示：务必按 A 键关闭环境光源，这样该演示程序看起来将非常酷。

12.5.3　实现完美透视修正纹理映射

现在，要实现完美（perfect）透视修正纹理映射，只需修改一些代码即可。我们只需以使用 1/z 缓存的仿射纹理映射函数为基础，对插值代码以及计算每个像素的纹理坐标 u 和 z 的代码做些修改。需要修改的地方如下：

- 在初始化阶段，需要计算 *u/z*、*v/z* 和 *1/z*。
- 在光栅化函数的内循环中，需要计算纹理坐标 *u* 和 *v*。这是通过将 *u/z* 和 *v/z* 除以 1/z 得到透视修正纹理坐标 *u* 和 *v* 来实现的，这些坐标将用于从纹理图中采集样本。

当然，魔鬼隐藏在细节之中。首先，需要再谈一下定点格式。现在，通过插值计算的不是 *u* 和 *v*，而是 *u/z* 和 *v/z*，因此要求小数部分的精度高于 16 位（二进制）。然而，不能像 1/z 那样使用 4.28 格式，因为纹理坐标最大可达 255。要表示 0～255 之间的值，至少需要 8 位，加上一个符号位，总共需要 9 位。作者不喜欢奇数，因此使用 10 位来存储整数部分，余下的 22 位用于存储小数部分。

为插值计算 *u/z* 和 *v/z*，将使用 10.22 定点格式；为插值计算 *1/z*，将使用 4.28 定点格式。这意味着将 *u/z* 和 *v/z* 除以 *1/z* 时，是用一个 4.28 格式的定点数去除以一个 10.22 格式的定点数，因此一定要注意执行正确的移位运算。

来看如何设置顶点和 *u*、*v* 信息。下面的代码摘自支持 1/z 缓存、透视修正和固定着色的光栅化函数：

```
// 提取顶点信息
x0 = (int)(face->tvlist[v0].x+0.0);
y0 = (int)(face->tvlist[v0].y+0.0);

tu0 = ((int)(face->tvlist[v0].u0+0.5) << FIXP22_SHIFT) /
      (int)(face->tvlist[v0].z+0.5);
tv0 = ((int)(face->tvlist[v0].v0+0.5) << FIXP22_SHIFT) /
      (int)(face->tvlist[v0].z+0.5);

tz0 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v0].z+0.5);

x1 = (int)(face->tvlist[v1].x+0.0);
y1 = (int)(face->tvlist[v1].y+0.0);

tu1 = ((int)(face->tvlist[v1].u0+0.5) << FIXP22_SHIFT) /
    (int)(face->tvlist[v1].z+0.5);
tv1 = ((int)(face->tvlist[v1].v0+0.5) << FIXP22_SHIFT) /
    (int)(face->tvlist[v1].z+0.5);

tz1 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v1].z+0.5);

x2 = (int)(face->tvlist[v2].x+0.0);
y2 = (int)(face->tvlist[v2].y+0.0);
tu2 = ((int)(face->tvlist[v2].u0+0.5) << FIXP22_SHIFT) /
    (int)(face->tvlist[v2].z+0.5);
tv2 = ((int)(face->tvlist[v2].v0+0.5) << FIXP22_SHIFT) /
    (int)(face->tvlist[v2].z+0.5);

tz2 = (1 << FIXP28_SHIFT) / (int)(face->tvlist[v2].z+0.5);
```

常量 FIXP22_SHIFT 的定义如下：

```
#define FIXP22_SHIFT 22
```

请读者花些时间来弄懂这些设置代码。对 1/z 的计算与前面相同，同时将每个顶点的纹理坐标设置为 *u/z* 和 *v/z*。首先将 *u* 和 *v* 进行移位，将其转换为 10.22 定点格式，然后将其除以 *z*。正如本章前面证明的，

将定点数除以或乘以整数时，结果为定点数，当然条件是不发生上溢或下溢。经过这种初始化后，便可以通过线性插值来计算 1/*z*、*u*/*z* 和 *v*/*z* 了。

除了最内面的循环外，光栅化函数的代码与以前相同。在内循环中，需要使用纹理坐标 *u* 和 *v* 来访问纹理内存，但为支持完美的透视修正纹理映射，需要将 *u*/*z* 和 *v*/*z* 分别除以 1/*z* 来计算它们：

```
u = (u/z) / (1/z)
v = (v/z) / (1/z)
```

然而，u/z 和 v/z 为 10.22 定点格式，而 1/z 为 4.28 定点格式，因此它们不能直接相除，而必须在执行除法运算前对一个或两个操作数进行移位，这样得到的 u 和 v 将为简单整型格式，且不会降低精度。下面是绘制扫描线的内循环代码：

```
for (xi=xstart; xi < xend; xi++)
  {
  // 检测当前像素的 1/z 值是否比 1/z 缓存中相应的值大
  if (zi > z_ptr[xi])
    {
    // 绘制像素
    screen_ptr[xi] = textmap[ ((ui << (FIXP28_SHIFT - FIXP22_SHIFT)) / zi) +
   ( ((vi << (FIXP28_SHIFT - FIXP22_SHIFT)) / zi)  << texture_shift2)];

    // 更新 1/z 缓存
    z_ptr[xi] = zi;
    } // end if
    // 计算下一个像素的 u、v、z
    ui+=du;
    vi+=dv;
    zi+=dz;
    } // end for xi
```

其中计算 *u* 和 *v* 的代码为粗体。使用仿射纹理映射时，相应的代码如下：

```
screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
          ((vi >> FIXP16_SHIFT) << texture_shift2)];
```

现在来看除以 z 和移位的问题。为何左移位数为 FIXP28_SHIFT – FIXP22_SHIFT = 6 位呢？请看图 12.22。

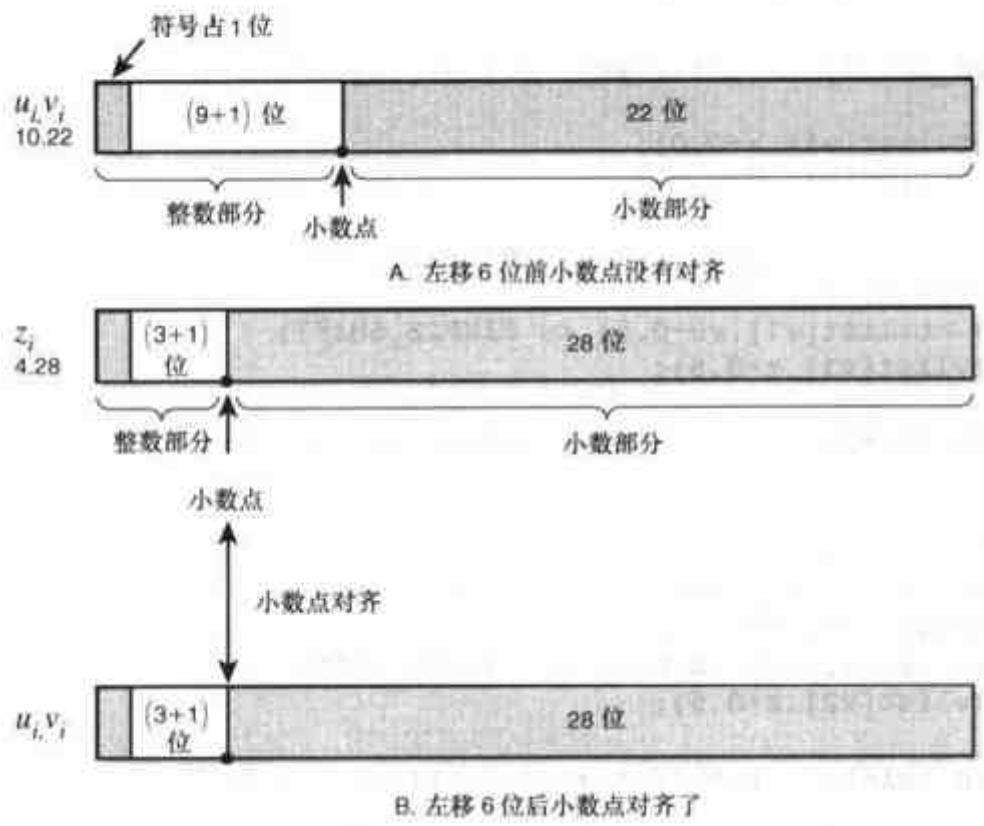


图 12.22　使小数点对齐

ui 和 *vi* 都是 10.22 格式，而 *zi* 为 4.28 格式。要使除法运算的结果没有被缩放且为整数，需要将小数点对齐。为此，需要将 *ui* 和 *vi* 左移 6 位，以便与 *zi* 对齐。这是因为在这两种格式中，小数部分相差的位数为：

```
(FIXP28_SHIFT - FIXP22_SHIFT) = (28 - 22) = 6
```

通过移位，被除数和除数都是 4.28 格式。这样，得到的商将为整数，且没有被缩放。

提示：对于纹理寻址，还存在另一种优化方法。访问纹理的代码如下：

```
texture[u + (v << n)]
```

其中 *n* 为纹理宽度的以 2 为底的对数，即每行占据的字数。上述移位相当于一次乘法运算。如果预先将 *vi* 放大纹理宽度倍，则可避免每次计算纹理地址时都执行左移运算，从而将计算纹理地址的速度提高大约 30%。这种优化将在以后再去实现。

有关透视修正纹理映射就介绍到这里。虽然对于每个像素增加两次除法运算并非不可接受的，但稍后将通过多种方法解决这种问题。至少，通过计算除数的倒数，可以将除法转换为乘法。例如：

```
float x,y,c;

// 使用 x、y、c 执行一些计算

x = x/c;
y = y/c;
```

假设每次除法运算需要 15～40 个时钟周期，执行上述代码时将需要 30～80 时钟周期。现在对其做简单的变换：

```
float x,y,c;

// 使用 x、y、c 执行一些计算

float oneoverc = 1/c;

x = x*c;
y = y*c;
```

这包含一次除法运算（需要 15～40 个时钟周期）和两次乘法运算（各需 2～3 个时钟周期），总共需要 34～46 个时钟周期。虽然这里对所需周期数的估计太过悲观了，但通过首先计算倒数，可节省大约 50%的时钟周期。在最内面循环中，使用除法运算都是不能接受的，必须对其进行优化。在算法方面，惟一可能的优化是进行近似，这将稍后讨论。现在来看一下支持 1/*z* 缓存和透视修正纹理映射的函数。

```
void Draw_Textured_Perspective_Triangle_INVZB_Alpha16(
                  POLYF4DV2_PTR face, // 指向多边形面的指针
                  UCHAR *_dest_buffer, // 指向视频缓存的指针
                  int mem_pitch,    // 每行多少字节（320、640 等）
                  UCHAR *_zbuffer,   // 指向 z 缓存的指针
                  int zpitch,      // z 缓存中每行多少字节
                  int alpha);      // alpha 等级
```

该函数使用透视修正纹理映射、alpha 混合和固定着色来绘制三角形。

```
void Draw_Textured_Perspective_Triangle_FSINVZB_Alpha16(
           POLYF4DV2_PTR face, // 指向多边形面的指针
           UCHAR *_dest_buffer, // 指向视频缓存的指针
           int mem_pitch,    // 每行多少字节（320、640 等）
           UCHAR *_zbuffer,   // 指向 z 缓存的指针
           int zpitch,      // z 缓存中每行多少字节
           int alpha);      // alpha 等级
```

该函数使用透视修正纹理映射、alpha 混合和恒定着色来绘制三角形。

```
void Draw_Textured_Perspective_Triangle_INVZB_16(
           POLYF4DV2_PTR face, // 指向多边形面的指针
           UCHAR *_dest_buffer, // 指向视频缓存的指针
           int mem_pitch,     // 每行多少字节（320、640 等）
           UCHAR *_zbuffer,   // 指向 z 缓存的指针
           int zpitch);       // z 缓存中每行多少字节
```

该函数使用透视修正纹理映射和固定着色来绘制三角形。

```
void Draw_Textured_Perspective_Triangle_FSINVZB_16(
            POLYF4DV2_PTR face,// 指向多边形面的指针
            UCHAR *_dest_buffer,// 指向视频缓存的指针
            int mem_pitch,   // 每行多少字节（320、640 等）
            UCHAR *_zbuffer,  // 指向 z 缓存的指针
            int zpitch);     // z 缓存中每行多少字节
```

该函数使用透视修正纹理映射和恒定着色来绘制三角形。

为演示透视修正纹理映射，最佳的选择当然是使用室内环境，但我们还没有创建室内环境的有效工具。DEMOII12_6.CPP|EXE 创建了一个非常平的地形，并将一个棋盘纹理映射到地形的上面。这个演示程序表明，仿射纹理映射和透视修正纹理映射之间有天壤之别。图 12.23 是该演示程序运行时的屏幕截图。

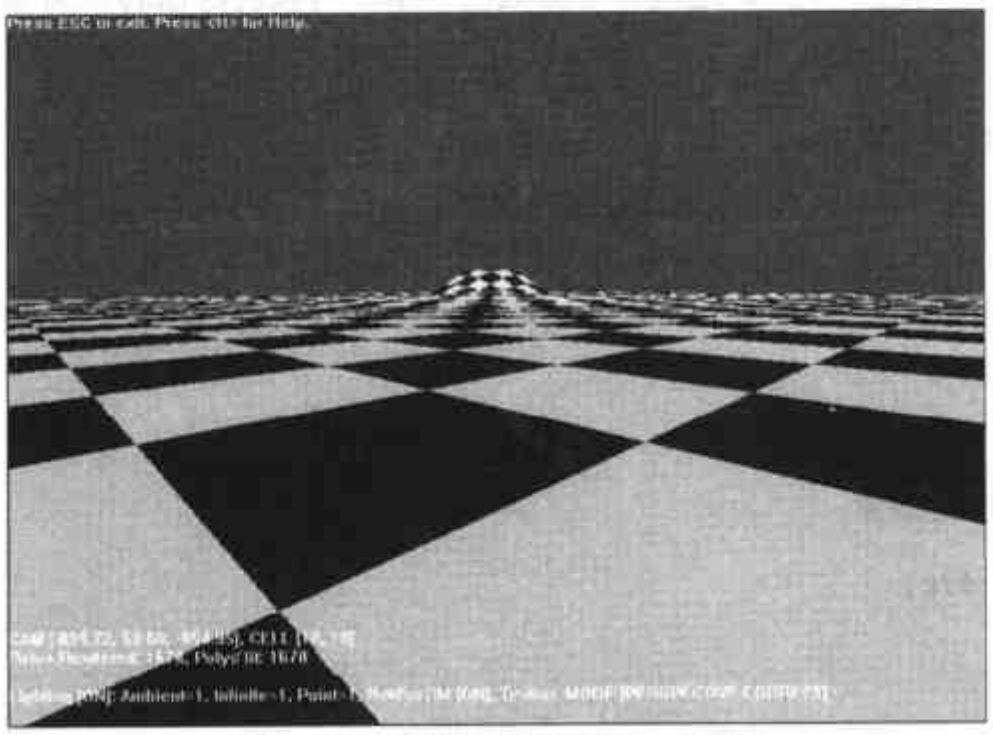


图 12.23　透视修正纹理映射演示程序

和以往一样，按箭头键可在地形上漫游，按 T 键可在仿射纹理映射和透视修正纹理映射之间切换。要编译该演示程序，需要 DEMOII12_6.CPP|H、T3DLIB1-10.CPP|H 和 DirectX .LIB 文件。

12.5.4　实现线性分段透视修正纹理映射

虽然可以通过先计算倒数的方式对两次除法运算进行优化，以提高完美纹理映射函数的速度，但除法运算毕竟是除法运算，在大多数情况下，对软件引擎来说这是不能接受的。那么还有其他方法可以提高其速度吗？可以通过近似来提高速度，同时确保视觉效果令人满意吗？近似都是基于误差在可接受范围内这样一个概念的，因此这完全取决于在最后渲染的图像中多大的误差是可以接受的。例如，对于纹理扭曲，第一人称射击游戏和室外地形游戏的容忍程度可能不一样，因此需要考虑各种各样的因素。可以实现一种这样的近似，我们可以对误差范围进行控制，使其在完全仿射纹理映射和完美透视修正纹理映射之间变化。不幸的是，本书中的实现没有这种功能，但如果需要，读者可以添加。

我们将实现的近似叫做线性分段近似（或分段线性近似，这取决于您站在什么角度）。基本上，它是仿射纹理映射和透视修正纹理映射之间的折衷，如图 12.24 所示。

不同于完美透视修正纹理映射，线性分段纹理映射每隔一定间隔（如 16～32 像素）计算像素的准确纹理坐标，然后在这些像素之间进行仿射线性纹理映射。图 12.25 是透视修正采样曲线和线性分段近似的采样曲线。

基本上，我们计算透视修正曲线上的纹理坐标，然后在它们之间进行线性插值。随着采样点数量的不断增多，线性分段近似曲线与透视修正采样曲线将越来越接近，这种技术非常适合用于按需纹理映射。通过控制质量和精度，可避免对每个像素都执行除法运算，而是每隔 n 个像素执行这些除法运算，然后通过线性插值计算这些像素之间的纹理坐标。当物体离视平面较近时，提高近似的精度（增加采样点），当多边形远离视平面时，减少采样点数目，几乎采用仿射纹理映射。

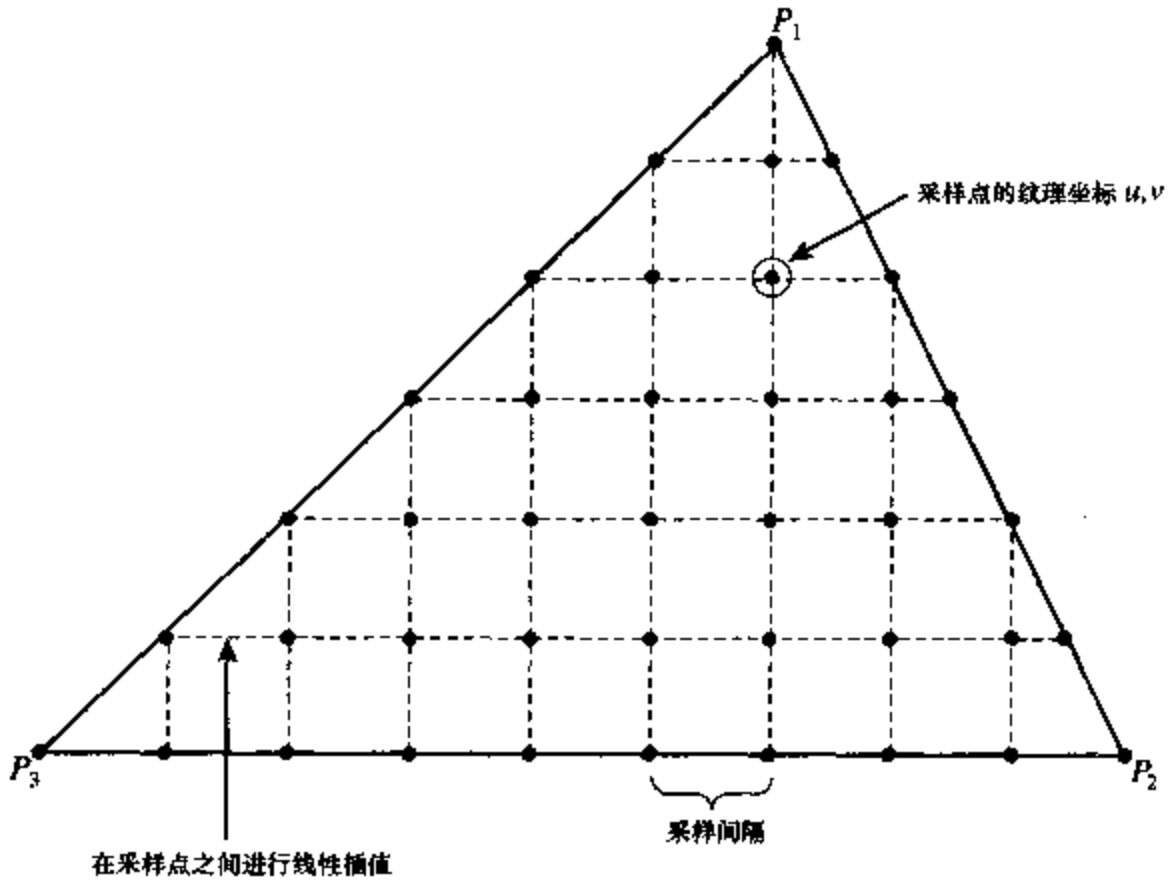


图 12.24　三角形中的线性分段近似采样点

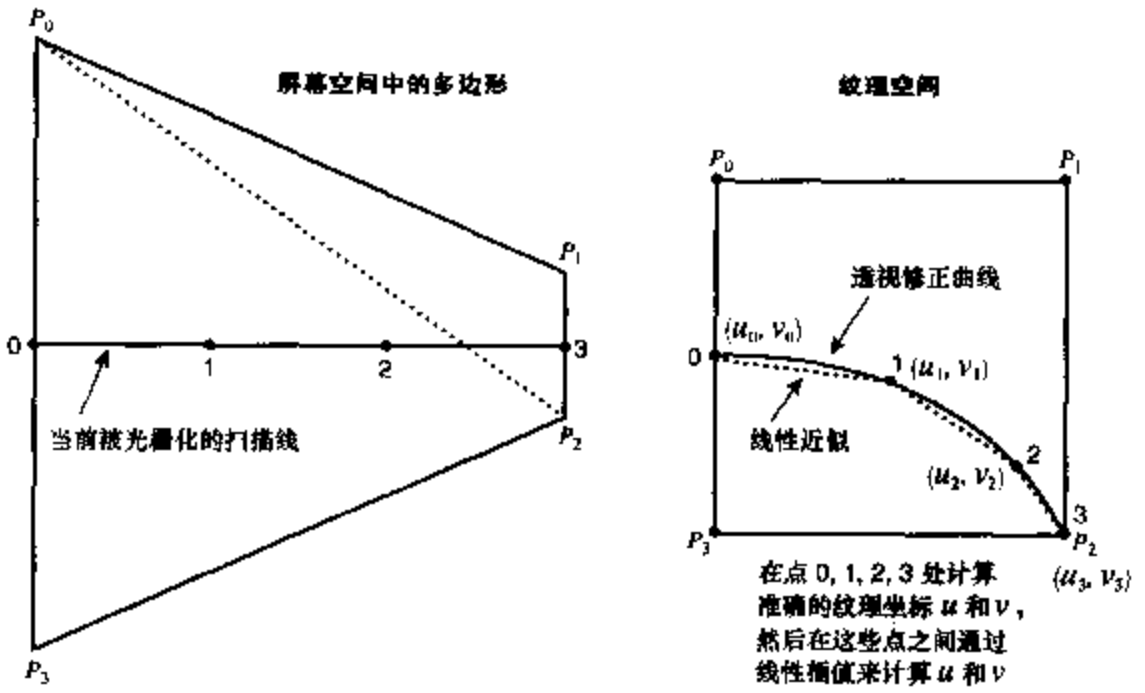


图 12.25　线性分段近似曲线和透视修正曲线

实际上，作者发现每隔 16～32 个像素计算准确的纹理坐标时，效果几乎与完美透视修正纹理映射相同，即使是在非常近的地方观察。不好的消息时，这种算法实现起来有些棘手，甚至令人讨厌。编写通用的每隔 n 个像素采样一次的代码是一项比较困难的工作。作者的意思是说，实现分段采样的方法有多种，如图 12.26 所示。

- **方法 1**：每个 n 条扫描线计算一次端点的准确纹理坐标，并在每条扫描线上每隔 n 个像素计算一次准确的纹理坐标。
- **方法 2**：计算每条扫描线端点的准确纹理坐标，并在每条扫描线上每隔 n 个像素计算一次准确的纹理坐标。
- **方法 3**：计算每条扫描线端点的准确纹理坐标，并在每条扫描线上通过线性插值计算纹理坐标。

现在我们实现方法 3，以后可能实现方法 2。原因在于，在刚编写的完美纹理映射函数中包含了计算每条扫描线端点的准确纹理坐标的逻辑，我们只需删除对每个像素计算准确纹理坐标的代码，并根据端点的准确纹理坐标 u 和 v 在每条扫描线上进行线性插值即可。可以实现方法 2，但在每条扫描线上每隔 n 个像素计算一次准确的 u、v 坐标的开销很大，当前还不需要这么高的精度；以后在模型和引擎中涉及室内环境时，如果需要将实现这种方法。

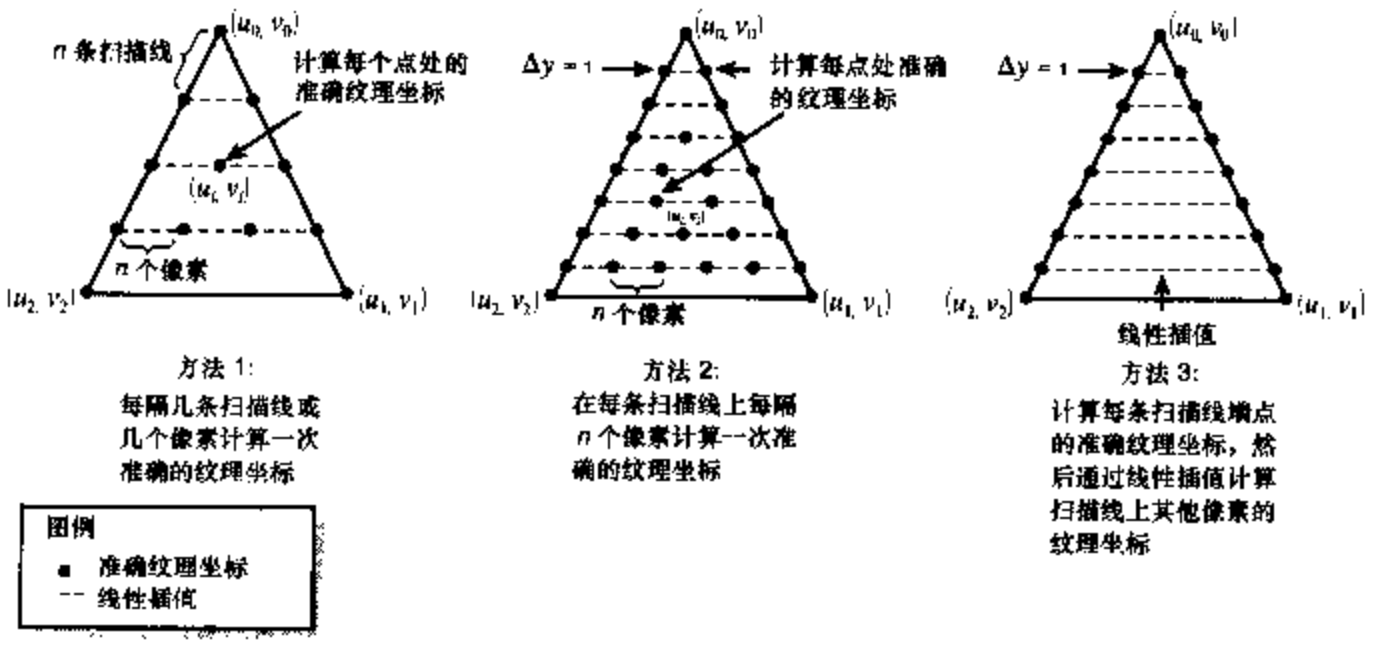


图 12.26　分段线性纹理映射中计算纹理坐标的方法

我们将在透视修正纹理映射函数的基础上进行修改：只计算每条扫描线端点的准确纹理坐标，而不计算每个像素的准确纹理坐标；然后根据端点的准确纹理坐标，通过线性插值从左到右依次计算扫描线上每个像素的纹理坐标。

现在来看看实现代码。下面的代码摘自最基本的分段线性纹理映射函数 Draw_Textured_PrespectiveLP_Triangle_INVZB_16()，它是渲染整个三角形的主循环：

```
// 将指针 screen_ptr 指向起始行
screen_ptr = dest_buffer + (ystart * mem_pitch);

// 将指针 z_ptr 指向起始行
z_ptr = zbuffer + (ystart * zpitch);

for (yi = ystart; yi < yend; yi++)
  {
  // 计算扫描线端点的 x 坐标
  xstart = ((xl + FIXP16_ROUND_UP) >> FIXP16_SHIFT);
  xend  = ((xr + FIXP16_ROUND_UP) >> FIXP16_SHIFT);

  // 计算 ul、ur、vl、vr
  ul2 = ((ul << (FIXP28_SHIFT - FIXP22_SHIFT)) / (zl >> 6) ) << 16;
  ur2 = ((ur << (FIXP28_SHIFT - FIXP22_SHIFT)) / (zr >> 6) ) << 16;

  vl2 = ((vl << (FIXP28_SHIFT - FIXP22_SHIFT)) / (zl >> 6) ) << 16;
  vr2 = ((vr << (FIXP28_SHIFT - FIXP22_SHIFT)) / (zr >> 6) ) << 16;

  // 计算 u、v、z 的初始值
  zi = zl + 0;
  ui = ul2 + 0;
  vi = vl2 + 0;

  // 计算 u、v 增量
  if ((dx = (xend - xstart))>0)
   {
   du = (ur2 - ul2) / dx;
   dv = (vr2 - vl2) / dx;
   dz = (zr - zl) / dx;
   } // end if
  else
   {
   du = (ur2 - ul2) ;
   dv = (vr2 - vl2) ;
   dz = (zr - zl);
   } // end else

  // 绘制扫描线
  for (xi=xstart; xi < xend; xi++)
```

```
{
  // 检查当前像素的 z 值是否大于 z 缓存中相应的值
  if (zi > z_ptr[xi])
    {
    // 绘制像素
    screen_ptr[xi] = textmap[(ui >> FIXP22_SHIFT) +
                ((vi >> FIXP22_SHIFT) << texture_shift2)];

    // 更新 z 缓存
    z_ptr[xi] = zi;
    } // end if
    // 计算下一个像素的 u、v、z
    ui+=du;
    vi+=dv;
    zi+=dz;
    } // end for xi

    // 计算下一条扫描线端点的 u、v、z
    xl+=dxdyl;
    ul+=dudyl;
    vl+=dvdyl;
    zl+=dzdyl;

    xr+=dxdyr;
    ur+=dudyr;
    vr+=dvdyr;
    zr+=dzdyr;

    // 将指针 screen_ptr 指向下一行
    screen_ptr+=mem_pitch;

      // 将指针 z_ptr 指向下一行
      z_ptr+=zpitch;

    } // end for y
```

这些代码脱离了上下文，有些难以理解；但请花些时间慢慢地啃。其中两段重要的代码为粗体。第一段通过除以 1/*z* 计算当前扫描线端点的准确 *u*、*v* 坐标，结果存储在 *ul*2、*vl*2、*ur*2 和 *vr*2 中。第二段是扫描线光栅化代码中最内面的循环，它根据 *ul*2、*vl*2、*ur*2 和 *vr*2，通过线性插值来计算纹理坐标。虽然仍通过除法运算来计算准确的纹理坐标 *u* 和 *v*，但现在只是针对每条扫描线而不是每个像素执行这种运算。另外，在每条扫描线上每隔 *n* 个像素计算一次准确的纹理坐标并不是那么复杂，但也不是那么简单，而且会使内循环非常混乱，因此暂时不去管它。

有关分段线性透视修正纹理映射就介绍到这里，它只是有选择地计算某些像素的准确纹理坐标，然后在这些像素之间通过线性插值计算纹理坐标。下面来看一下支持这种功能的函数（它们都使用 1/*z* 缓存）：

```
void Draw_Textured_PerspectiveLP_Triangle_FSINVZB_16(
          POLYF4DV2_PTR face,// 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,    // 每行多少字节（320、640 等）
          UCHAR *_zbuffer,   // 指向 z 缓存的指针
          int zpitch);     // z 缓存中每行多少字节
```

该函数支持恒定着色处理和线性分段透视修正纹理映射。

```
void Draw_Textured_PerspectiveLP_Triangle_INVZB_16(
        POLYF4DV2_PTR face, // 指向多边形面的指针
        UCHAR *_dest_buffer, // 指向视频缓存的指针
        int mem_pitch,    // 每行多少字节（320、640 等）
        UCHAR *_zbuffer,   // 指向 z 缓存的指针
        int zpitch);     // z 缓存中每行多少字节
```

该函数支持固定着色和线性分段透视修正纹理映射。

```
void Draw_Textured_PerspectiveLP_Triangle_FSINVZB_Alpha16(
          POLYF4DV2_PTR face, // 指向多边形面的指针
```

```
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,    // 每行多少字节（320、640 等）
            UCHAR *_zbuffer,   // 指向 z 缓存的指针
            int zpitch,     // z 缓存中每行多少字节
            int alpha);     // the alpha
```

该函数支持恒定着色、线性分段透视修正纹理映射和 alpha 混合。

```
void Draw_Textured_PerspectiveLP_Triangle_INVZB_Alpha16(
            POLYF4DV2_PTR face, // 指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,    // 每行多少字节（320、640 等）
            UCHAR *_zbuffer,   // 指向 z 缓存的指针
            int zpitch,     // z 缓存中每行多少字节
            int alpha);     // alpha
```

该函数支持固定着色、线性分段透视修正纹理映射和 alpha 混合。

为比较仿射纹理映射、线性分段近似纹理映射和完美透视修正纹理映射，作者对前一个演示程序进行了更新，使之支持线性分段光栅化函数。图 12.27 是 DEMOII12_7.CPP|EXE 运行时的屏幕截图。

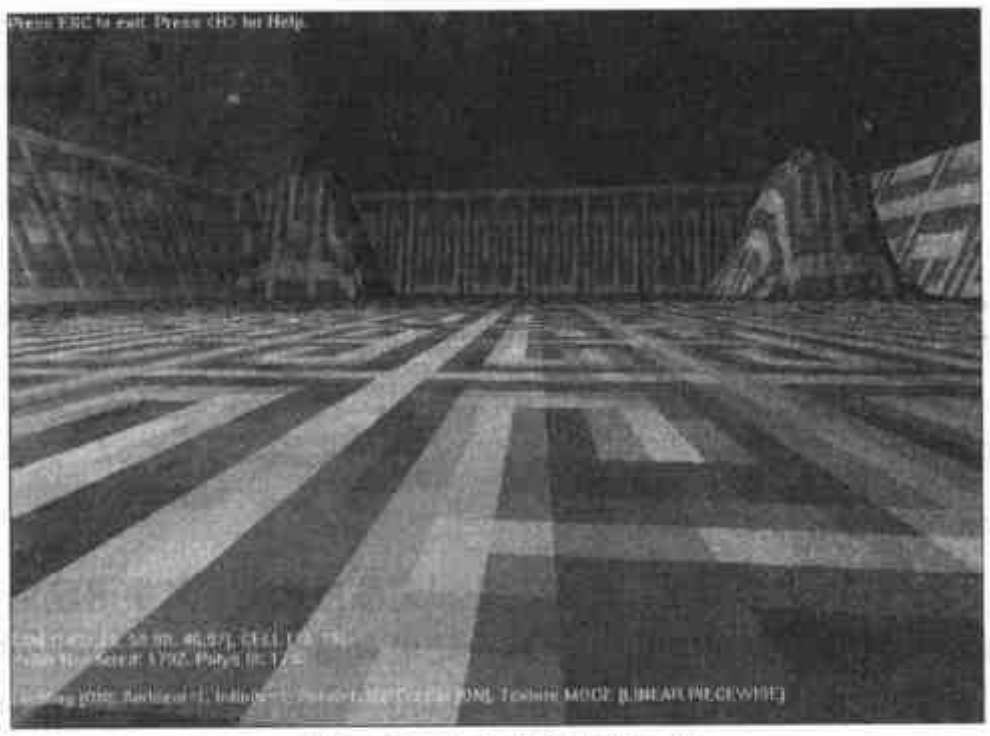


图 12.27　线性分段纹理映射

线性分段纹理映射的效果看起来不太糟糕。该演示程序的控制方法与以前相同：按箭头键移动；按 T 键在仿射、线性分段和完美透视修正纹理映射之间进行切换。

12.5.5　透视修正纹理映射的二次近似

作者要讨论的下一种近似技术叫二次近似（quadratic approximation）。它基于线性分段近似的思想，但不是在每条扫描线上进行线性插值，而是用二次曲线来逼近透视曲线。读者可能认为，这比对每个像素执行两次除法运算更糟，但并非如此。事实上，这几乎与线性分段一样容易，但效果几乎与完美透视修正纹理映射相同。这里不打算实现它，因为还不需要；但如果读者愿意可以实现它。然而，我们将介绍有关其工作原理的每个细节，事实上它非常简单。

首先来定义问题。假设仍计算每条扫描线端点的准确纹理坐标（可能通过乘以倒数来进行除法运算）。我们希望在每条扫描线上进行比线性插值更准确的近似，但又不想采用将扫描线分成多段的方法。解决方案是，在扫描线上从左到右进行插值，但不采用线性插值，而是使用一条通过透视曲线上三个点的曲线，或至少使用一条比线性插值使用的直线更接近于透视曲线的曲线。图 12.28 说明了这一点。

因此，我们需要一条形状与 *c/z* 类似的曲线，其中 *c* 为常量；这是二次曲线的用武之地。我们知道，下面这样的函数表示一条抛物线：

$$a*x^2 + bx + c = 0$$

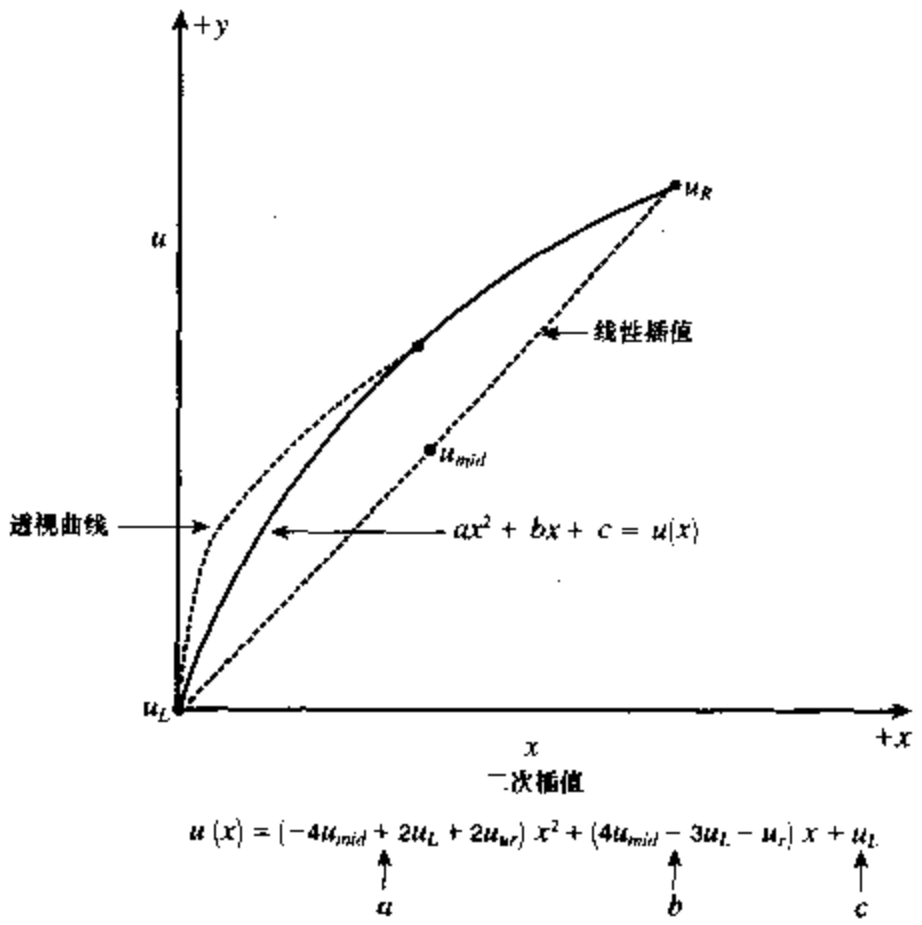
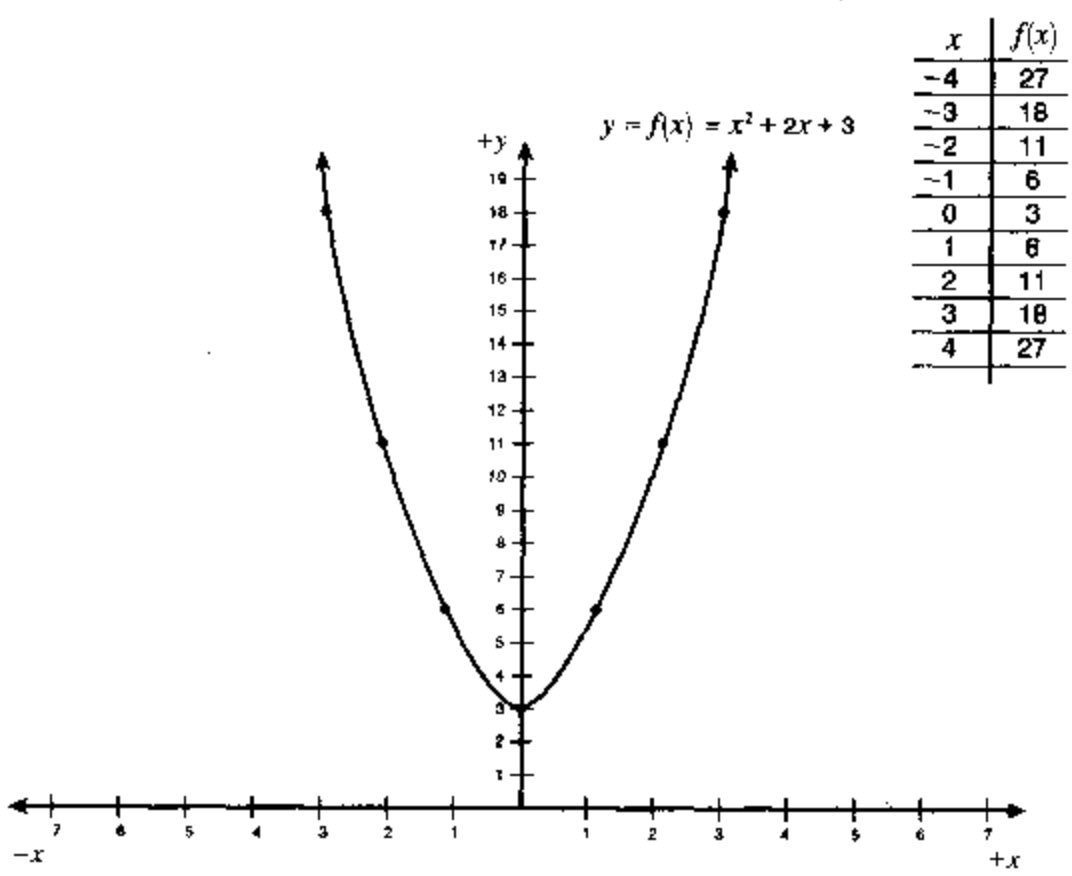


图 12.28　与透视曲线匹配的二次近似

例如，图 12.29 说明了下述曲线的外观：

f(x) = 1*x² + 2*x + 3



值表

x	f(x)
−4	27
−3	18
−2	11
−1	6
0	3
1	6
2	11
3	18
4	27

图 12.29　标准抛物线

我们需要有一种将通用二次曲线 $a*x^2 + b*x + c$ 映射到纹理曲线 $c/2$ 的方法。读者一定要弄懂这种方法，因为它是计算机图形学中很多近似的关键所在。另外，虽然二次曲线是抛物线，但仅当定义域为(−∞，+∞)时才这样。可以用二次曲线中定义域[i，j]对应的部分来近似。为此，需要计算出二次曲线的系数 a、b 和 c。

为此，需要确定系数 a、b 和 c，使二次曲线与透视曲线尽可能接近。我们将计算每个端点的纹理坐标，并使用它们来求解 a、b 和 c。这样，二次曲线将经过两个端点，但曲线中间呢？因此还需要其他的信息。有三个未知量（a、b 和 c），但只有两个已知量（ul 和 ur 或 vl 和 vr），它们是扫描线端点的纹理坐标。u 和 v 是独立的，必须分别进行插值，因此需要一条用于插值计算 u 的二次曲线和一条用于插值计算 v 的二次

曲线：但 u 和 v 的计算方法相同。

用作插值曲线的二次多项式如下：

$$f(x) = 1*x^2 + 2*x + 3$$

我们还知道两个端点的纹理坐标 ul 和 ur；但由于二次曲线有三个系数，因此还需要另一点的准确纹理坐标 u 和 v，才能计算出系数 a、b 和 c。我们已经知道扫描线两个端点的准确纹理坐标，还将计算中点的准确纹理坐标 u 和 v，如图 12.30 所示。

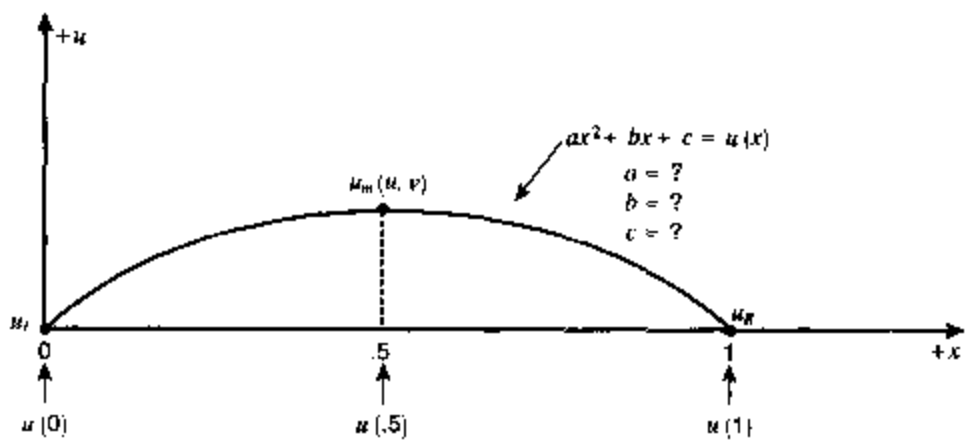


图 12.30　根据端点和中点计算二次曲线的系数

要计算中点的 u 和 v，需要执行两次除法运算，这是不可避免的。为此，需要将当前扫描线的 u/z、v/z 增量（$(u_z)_i$和$(v_z)_i$）乘以 0.5 和 dx；还必须计算中点的 $1/z$，用它来除以中点的 u/z 和 v/z，以得到中点的 u 和 v。因此需要将当前扫描线的 $1/z$ 增量（$(1\_z)_i$）乘以 0.5 和 dx；然后将这些值分别与 ul、vl 和 zl（左端点的 u/z、v/z 和 $1/z$）相加，然后使用除法来计算中点的 u 和 v，如图 12.30 所示。

给定下列信息：

- u_zl、v_zl 和 zl——当前扫描线左端点的 u/z、v/z 和 $1/z$；
- $(1\_z)_i$——当前扫描线的 $1/z$ 增量；
- $(u\_z)_i$——当前扫描线的 u/z 增量；
- $(v\_z)_i$——当前扫描线的 v/z 增量；
- dx——当前扫描线左右端点之间的距离，等于 $x_r - x_l$。

我们需要计算中点的 u 和 v。

首先来计算中点的 1/z，因为需要用它来除以中点的 u/z 和 v/z，以得到 u 和 v：

$$1\_z_{mid} = 1/zl + [(1\_z)_i * (0.5*dx)]$$

有了中点的 1/z 后，计算中点的 u 和 v：

$$u_{mid} = u\_zl + [(u\_z)_i * (0.5*dx)] \;/\; 1\_z_{mid}$$

$$v_{mid} = v\_zl + [(v\_z)_i * (0.5*dx)] \;/\; 1\_z_{mid}$$

虽然解释如何计算中点的 u 和 v 犹如解释麦田里的怪圈一样难，但具体计算却很简单。我们只需要该死的中点 u 和 v！有了它们后，回到二次近似。我们需要计算 u 插值曲线和 v 插值曲线的系数，但计算方法相同，因此只需推导出 u 插值曲线系数的计算方法，便可以使用相同的方法来计算 v 插值曲线的系数。另外，在不失普遍性的情况下，我们假设扫描线起点和终点的 x 坐标为 0 和 1（而不是 xl 和 xr）来计算系数。

给定扫描线左端点的纹理坐标 u 为 ul，右端点为 ur，中点为 u_{mid}，需要计算下述二次曲线的系数 a0、b0 和 c0：

$$f(x) = a0*x^2 + b0*x + c0$$

将（0, ul）和（1, ur）代入上述方程，可得如下结果：

$$f(0) = a0*0^2 + b0*0 + c0 = ul$$
$$c0 = ul$$
$$f(1) = a0*1^2 + b0*1 + c0 = ur$$
$$a0 + b0 + c0 = ur$$

将 $c0 = ul$ 代入最后一个等式，结果为：

```
a0 + b0 + ul = ur
a0 + b0 = (ur - ul)
```

将（0.5, u_{mid}）以及 c0 = *ul* 代入前面的方程，结果为：

```
 f(0.5)    = a0*0.52 + b0*0.5 + c0 = u_mid
a0*0.25 + b0*0.5 + ul = u_mid
a0*0.25 + b0*0.5 = (u_mid - ul)
```

这样，我们将有下列两个方程，可根据它们解出 a0 和 b0：

```
(1)   * a0 + (1) * b0 = (ur - ul)
(0.25)* a0 + (0.5)* b0 = (u_mid - ul)
```

将第二个方程两边乘以 4，结果如下：

```
(1)* a0 + (1)* b0 = (ur - ul)
(1)* a0 + (2)* b0 = 4*(u_mid - ul)
```

将第二个方程减去第一个方程，结果如下：

```
(1)* a0 + (1)* b0 = (ur - ul)
     (1)* b0 = 4*(u_mid - ul) - (ur - ul)
 = (4*u_mid - 3*ul - ur)
```

因此 b0 = (4*u_{mid} – 3*ul – ur)。将 b0 代入第一个方程，结果如下：

```
a0 + (4*u_mid - 3*ul - ur) = (ur - ul)

a0    = (ur - ul) - (4*u_mid - 3*ul - ur)
   = ur - ul - 4*u_mid + 3*ul + ur
   = -4*u_mid + 2*ul + 2*ur
```

因此：

```
a0 = (-4*u_mid + 2*ul + 2*ur)
b0 = (4*u_mid - 3*ul - ur)
c0 = ul
```

所以 u(x)曲线方程如下：

```
u(x) = a0*x² + b0*x + c0
   = (-4*u_mid + 2*ul + 2*ur)*x² + (4*u_mid - 3*ul - ur)*x + ul
```

纹理坐标 v 的插值曲线与此相同（假设系数为 a1、b1 和 c1）：

```
a1 = (-4*v_mid + 2*vl + 2*vr)
b1 = (4*v_mid - 3*vl - vr)
c1 = vl
```

至此，我们有了纹理坐标 u、v 的插值曲线，更重要的是，现在计算每个像素的纹理坐标时不再需要执行除法运算。我们将两次除法运算换成了 5 次乘法运算和 4 次加法运算，因为现在需要对每个像素执行下列运算：

```
u(x) = a0*x² + b0*x + c0
v(x) = a1*x² + b1*x + c1
```

注意：可以一次性计算 u(x)和 v(x)中的 x^2。

使用前向差分来计算二次多项式的值

实际上，采用蛮干的方法来计算多项式的值时，速度仍比每个像素执行两次除法运算要快，但可以采用一些技巧来进一步提高计算速度，如前向差分，因为插值曲线只有一个变量(x)。要计算前向差分，需要找出 u(x+1)和 u(x)之间的关系。我们假设：

```
u(x+1) = u(x) + du
```

其中 du 为前向差分，因此：

```
du = u(x+1) - u(x)
```

使用通用的二次曲线方程来计算 du：

$$
\begin{aligned}
du &= [a*(x+1)^2 + b*(x+1) + c] - [a*x^2 + b*x + c] \\
&= [a*x^2 + 2*a*x + a + b*x + b + c] - [a*x^2 + b*x + c] \\
&= a*x^2 + 2*a*x + a + b*x + b + c - a*x^2 - b*x - c \\
&= 2*a*x + a + b
\end{aligned}
$$

因此：

$$u(x+1) = u(x) + (2*a*x+a+b)$$

如果愿意，还可以进一步简化。现在，对于每个像素，计算一个纹理坐标时需要执行两次乘法运算和两次加法运算（如果将 2 和 a 合而为一，可再减少一次乘法运算），因此总共为 4 次乘法运算和 4 次加法运算。与需要 5 次乘法运算和 4 次加法运算的蛮干方法相比，这要好一些，但改进不大。下面对表示 du 的线性方程再进行前向差分。

假设其前向差分为 ddu，则：

$$du(x+1) = du(x) + ddu$$

因此：

$$ddu = du(x+1) - du(x)$$

代入 2*a*x + b + b，结果如下：

$$
\begin{aligned}
ddu &= [2*a*(x+1) + a + b] - [2*a*x + a + b] \\
&= 2*a*x + 2*a + a + b - 2*a*x - a - b \\
&= 2*a
\end{aligned}
$$

它是一个常量。现在，可以这样来计算 u(x+1)：

$$u(x+1) = u(x) + du$$

$$du = du(x) + ddu$$

我们只需首先计算出 u(0) 和 du(0)，然后使用前向差分来进行计算。这样，对于每个像素，插值计算一个纹理坐标需要执行两次加法运算，因此总共需要执行 4 次加法运算。

12.5.6 使用混合方法优化纹理映射

实现仿射、线性分段透视和完美透视纹理映射后，可根据要给什么东西贴上纹理、多边形离视平面的距离以及能接受的扭曲程度，调用不同的纹理映射函数来进行优化。例如，一种可能的方法是，对于离视平面很近的多边形，调用完美透视修正纹理映射函数；对于距离不远不近的多边形，调用线性分段纹理映射函数；对于较远的多边形，调用仿射纹理映射函数，如图 12.31 所示。

这样，即可以获得卓越的性能，又可以获得极好的视觉效果。实现这种优化非常简单。在渲染列表函数中，确定要渲染的多边形的 z 值，然后将其同不同的质量范围（完美、线性和仿射纹理映射）进行比较，并调用合适的纹理映射函数。其伪代码如下：

```
float perfect_range = 1000; // 距离为 0～1000 时使用完美透视修正纹理映射
float piecewise_range = 2000; // 距离为 1000～2000 时使用线性分段纹理映射
float affine_range = 2000;  // 距离大于 2000 时使用仿射纹理映射
// 选择光栅化函数
if (curr_poly->z < perfect_range)
  Perfect_Perspective_Texture_Mapper(curr_poly);
else
if (curr_poly->z > perfect_range && curr_poly->z < linear_range)
  Linear_Piecewise_Perspective_Texture_Mapper(curr_poly);
else
if (curr_poly->z > affine_range)
  Affine_Perspective_Texture_Mapper(curr_poly);
```

在实际的引擎中，函数名和访问顶点信息的方法与此不同，但重要的是这种算法。这确实是一种很好的优化方法，作者在最后的高级包装（wrapper）函数中实现了它，但只混合使用线性分段和仿射纹理映射函数，因

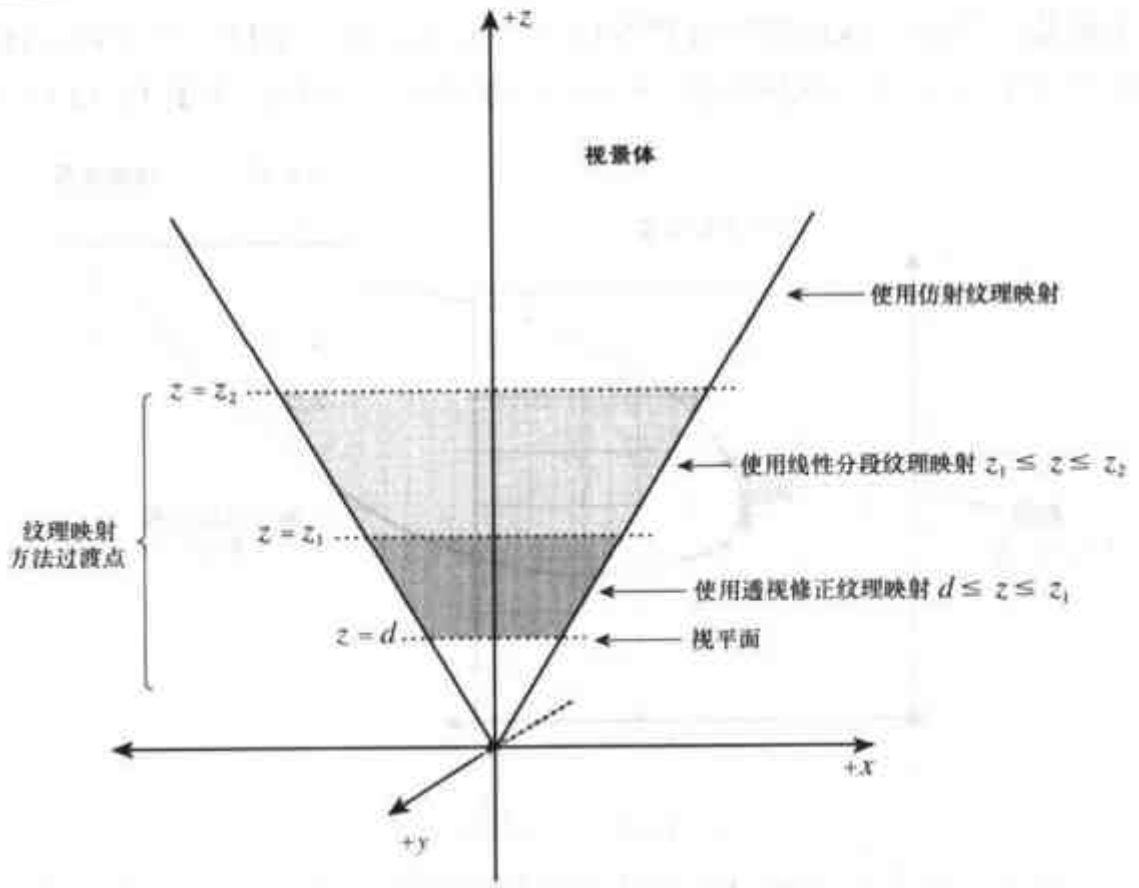


图 12.31　根据距离调用不同的纹理映射函数

为它们的速度快得多。这种选择可以通过编程实现，这样，调用高级函数时可选择优化方法（或不进行优化）。

为演示这种优化方法，作者手工实现了上述算法，因为我们还没有介绍包装函数；介绍该函数之前，需要讨论其他一些概念。该演示程序的文件名为 DEMOII12_8.CPP|EXE，图 12.32 是其运行时的屏幕截图。

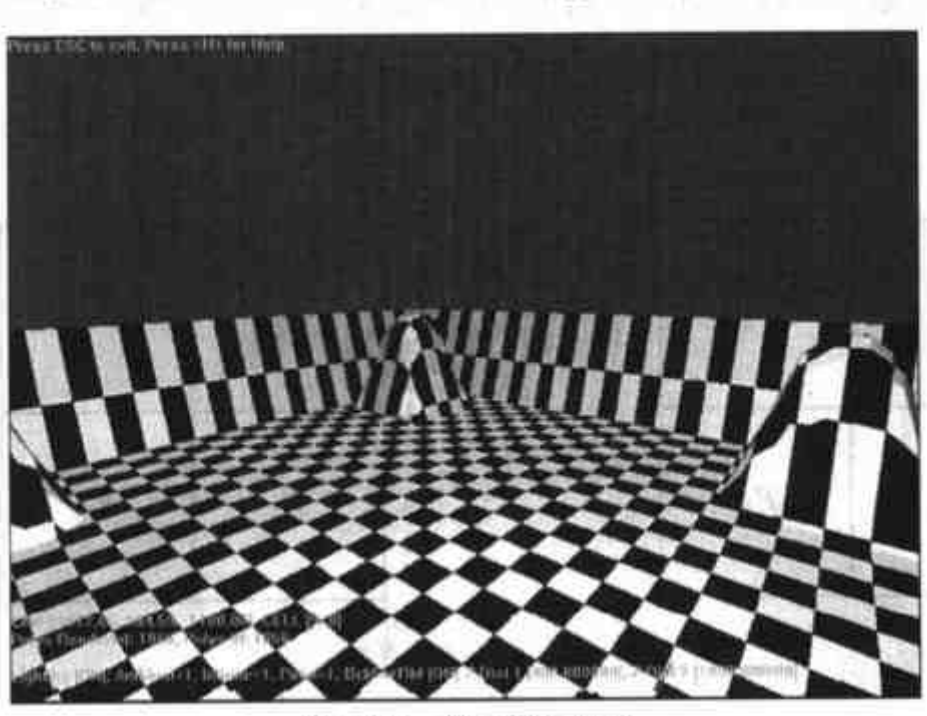


图 12.32　混合纹理映射

可以按下列键来修改使用完美、线性分段和仿射纹理映射的距离范围：

- 1：加大第一个过渡点的距离；
- 2：缩小第一个过渡点的距离；
- 3：加大第二个过渡点的距离；
- 4：缩小第二个过渡点的距离。

要编译这个程序，需要 DEMOII12_8.CPP|H、T3DLIB1-10.CPP|H 和 DirectX .LIB 文件。

12.6　双线性纹理滤波

双线性纹理滤波指的是在纹理映射期间，从源纹理中采集多个纹素样本，然后通过平均或使用其他滤

波器来计算目标像素的值。当前，我们进行纹理映射时，使用的是点采样，即计算纹理坐标 u 和 v，丢弃其小数部分（或进行四舍五入），然后根据它们来访问源纹理中的纹素，如图 12.33 所示。

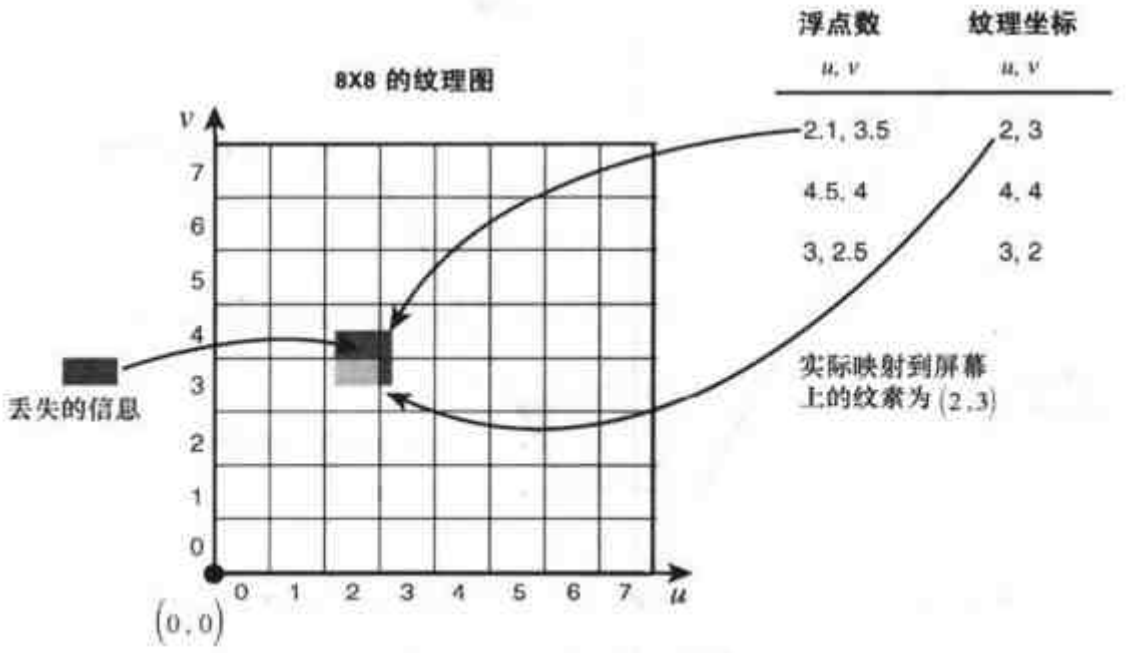


图 12.33　点采样机制

这样做的问题是，本来可用来绘制质量更高的图像的信息被丢弃了。这正是双线性插值（滤波）发挥作用的地方：不丢弃小数部分，而是根据纹理坐标 u、v 采集纹理图中相应点周围的像素，并以线性方式对它们进行平均或滤波。

例如，假设有一个 64×64 的纹理图，如图 12.34 所示，我们要采集周围的 4 个纹素，然后以线性方式（或使用线性插值和加权函数）计算它们的平均值。图 12.34 说明了这种处理方式。我们要根据周围的像素

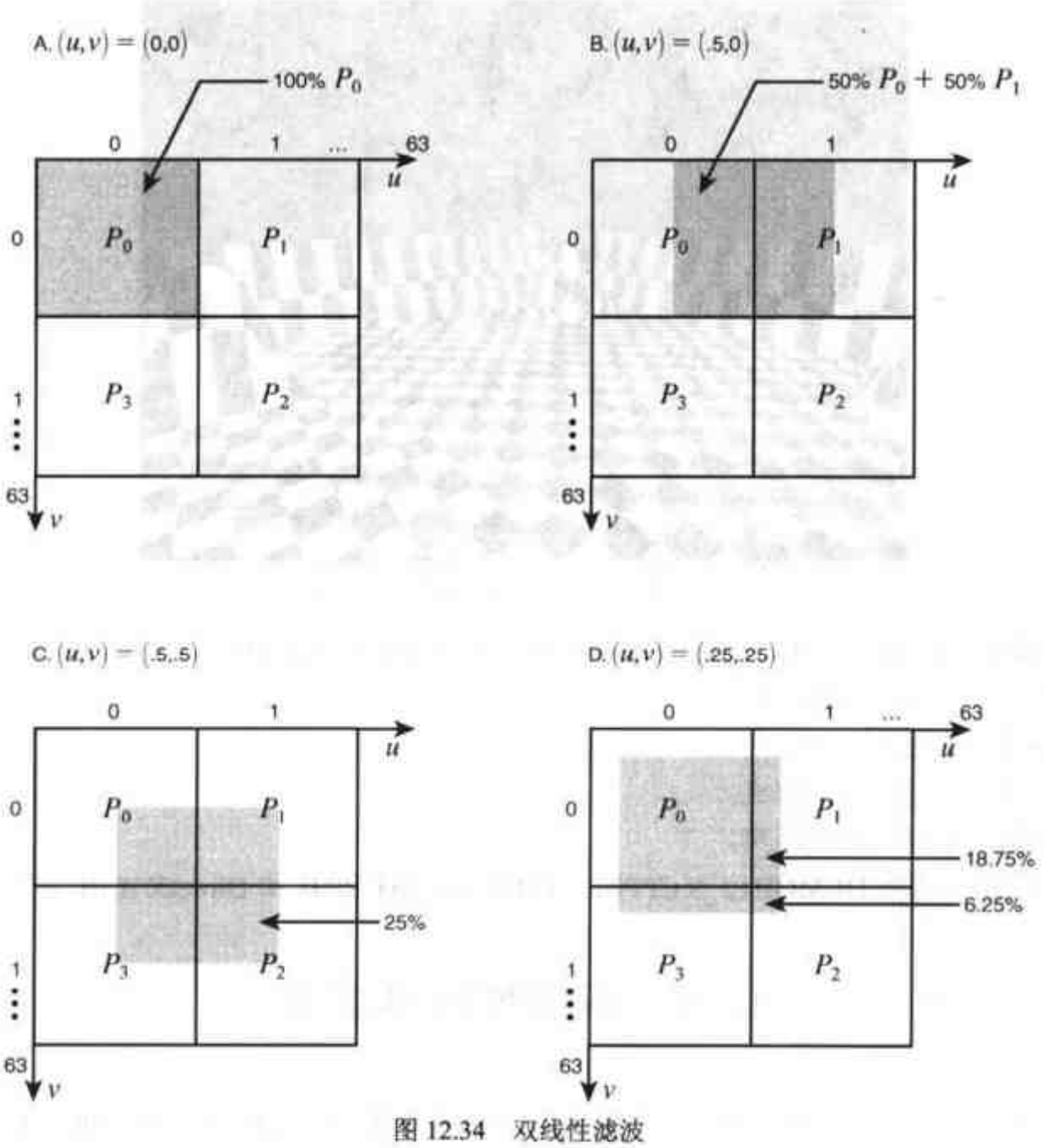


图 12.34　双线性滤波

p0、p1、p2 和 p3 来计算像素 pfinal 的值。可以这样考虑这个问题：pfinal 是一个采样框，它覆盖了一个由像素 p0～p3 组成的 2×2 像素阵列，然后根据对每个像素的覆盖率将像素 p0～p3 相加（当然是在 RGB 空间中）。图 12.34 包含几个范例，下面依次介绍它们。

在不失普遍性的情况下，我们假设像素阵列 p0～p3 位于纹理的左上角。接下来假设纹理坐标(u, v)为(0, 0)，如图 12.34a 所示。在这种情况下，采样框与像素(0, 0)重叠，因此 p0 占 100%，p1～p3 占 0%。所以，pfinal 等于 p0，用数学方法表示为：

$$p_{final} = (1.0)*p0 + (0.0)*p1 + (0.0)*p2 + (0.0)*p3$$

接下来看图 12.34b。采样框沿 u 轴移动，(u, v)为(0.5, 0)，即覆盖了像素 p0 和 p1 各 50%。因此，pfinal 为 50%的 p0 与 50%的 p1 之和，用数学方法表示为：

$$p_{final} = (0.5)*p0 + (0.5)*p1 + (0.0)*p2 + (0.0)*p3$$

下面来看图 12.34c，这是一种完整的 2D 双线性插值情形，纹理坐标(u, v)为(0.5, 0.5)，即采样框覆盖了每个纹素的 25%，用数学方法表示为：

$$\begin{aligned} p_{final} &= (0.5)*(0.5)*p0 + \\ &\quad (0.5)*(0.5)*p1 + \\ &\quad (0.5)*(0.5)*p2 + \\ &\quad (0.5)*(0.5)*p3 \\ &= (0.25)*p0 + (0.25)*p1 + (0.25)*p2 + (0.25)*p3 \end{aligned}$$

在这种情形下，不但要将每个像素样本乘以 u，还要乘以 v，因此为双线性插值。再来看图 12.34d，其中 u=0.25，v=0.25，用数学方法表示时，pfinal 如下：

$$\begin{aligned} p_{final} &= (1-0.25)*(1-0.25)*p0 + \\ &\quad (0.25)*(1-0.25)*p1 + \\ &\quad (0.25)*(0.25)*p2 + \\ &\quad (1-0.25)*(0.25)*p3 \\ &= (0.5625)*p0 + (0.1875)*p1 + (0.0625)*p2 + (0.1875)*p3 \end{aligned}$$

上述范例表明，只需将下述标准一维插值方法应用于二维空间即可：

```
(x)*value_1 + (1-x)*value_2
```

我们需要一个通用的公式，它让我们能够根据任何 u、v 值对纹理图进行采样，通过双线性插值计算得到最终的像素值。下面是在单色（而不是 RGB）情况下完成这项任务的伪代码：

```
// 单分量双线性纹理插值

// 单色纹理
float texture[TEXT_SIZE][TEXT_SIZE];

float u,v; // 纹理坐标，取值范围为 0 到 TEXT_SIZE - 1

// 计算 u 和 v 的整数部分
int ui = (int)u;
int vi = (int)v;

// 计算 u 和 v 的小数部分
float du = u - ui;
float dv = v - vi;

// 现在可以在纹理中进行采样了

float pfinal = (1-du)*(1-dv)* texture[u] [v]  +
       (du) *(1-dv)* texture[u+1][v]  +
       (du) *(dv) * texture[u+1][v+1] +
       (1-du)*(dv) * texture[u] [v+1];
```

正如读者看到的，双线性插值存在的问题是，需要执行大量的计算。在单色情况下，需要首先计算 u、

v 的整数部分和小数部分，然后计算 1 减去小数部分的值，这总共需要执行 4 次加法运算。接下来需要执行 8 次乘法运算，另外为计算 u+1 和 v+1，还需要执行两次加法运算。

因此，对于每个分量，需要执行 6 次加法运算和 8 次乘法运算。而每个像素有 3 个颜色分量，因此总共需要执行 18 次加法运算和 24 次乘法运算，对软件引擎而言，这样的计算量太大了。当然，通过使用一些技巧和优化，可以提高速度，但即使使用查找表，也需要执行大量的插值和数据提取运算。

因此，我们面临的问题是，实现双线性纹理映射/滤波是否值得？其速度相当慢，只能少量地使用，而不能将其应用于所有的多边形。如果想让少量的东西看上去非常美观，这样做是值得的。然而，通过使用模糊（blur）、高斯滤波器等删除高频元素，也可得到这样的效果。当然，这将降低纹理的分辨率，是一种折衷解决方案。

这里只在最基本的光栅化函数——不使用深度缓存、使用 z 缓存或使用 $1/z$ 缓存且支持固定着色和仿射纹理映射的光栅化函数（总共三个）——中实现双线性纹理映射。重新编写所有的纹理映射函数，使之支持双线性纹理映射是毫无意义的，原因是速度太慢，现在这样做不值得。另外，将至少介绍如何修改其中的一个纹理映射函数，让读者在需要时能够修改其他纹理映射函数。

这里介绍如何修改使用 z 缓存、支持固定着色和仿射纹理映射的函数。为此，以函数 Draw_Textured_TriangleZB2_16() 为基础，加入双线性纹理映射功能。下面是修改后的函数原型：

```
void Draw_Textured_Bilerp_TriangleZB_16(POLYF4DV2_PTR face, // 指向多边形面的指针
          UCHAR *_dest_buffer, // 指向视频缓存的指针
          int mem_pitch,     // 每行多少字节（320、640 等）
          UCHAR *_zbuffer,    // 指向 z 缓存的指针
         int zpitch)     // z 缓存中每行多少字节
```

该函数的调用方法与以前的版本相同，但它使用双线性纹理插值。在这个函数中，采用了前面的伪代码算法，但进行了大量的优化，同时使用了定点格式。下面是这个纹理映射函数的内循环代码：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
{
// 检测当前像素的 z 值是否小于 z 缓存中相应的值
if (zi < z_ptr[xi])
  {
  // 计算 u 和 v 的整数部分
  int uint = ui >> FIXP16_SHIFT;
  int vint = vi >> FIXP16_SHIFT;

  int uint_pls_1 = uint+1;
  if (uint_pls_1 > texture_size) uint_pls_1 = texture_size;

  int vint_pls_1 = vint+1;
  if (vint_pls_1 > texture_size) vint_pls_1 = texture_size;

  int texel00 = textmap[(uint+0)    + ((vint+0) << texture_shift2)];
  int texel10 = textmap[(uint_pls_1) + ((vint+0) << texture_shift2)];
  int texel01 = textmap[(uint+0)    + ((vint_pls_1) << texture_shift2)];
  int texel11 = textmap[(uint_pls_1) + ((vint_pls_1) << texture_shift2)];
  // 提取 R、G、B 分量
  int r_texel00 = ((texel00 >> 11)     );
  int g_texel00 = ((texel00 >> 5) & 0x3f);
  int b_texel00 = (texel00     & 0x1f);

  int r_texel10 = ((texel10 >> 11)     );
  int g_texel10 = ((texel10 >> 5) & 0x3f);
  int b_texel10 = (texel10     & 0x1f);

  int r_texel01 = ((texel01 >> 11)     );
  int g_texel01 = ((texel01 >> 5) & 0x3f);
  int b_texel01 = (texel01     & 0x1f);

  int r_texel11 = ((texel11 >> 11)     );
  int g_texel11 = ((texel11 >> 5) & 0x3f);
```

```
int b_texel11 = (texel11    & 0x1f);

// 计算 u 和 v (24.8 定点格式) 的小数部分
int dtu = (ui & (0xffff)) >> 8;
int dtv = (vi & (0xffff)) >> 8;

int one_minus_dtu = (1 << 8) - dtu;
int one_minus_dtv = (1 << 8) - dtv;

// 每个增量有 3 项组成: du、dv 和 texel
// 但计算 r_texel、g_texel 和 b_texel 时, du 和 dv 是通用的
// 因此只需计算它们一次
int one_minus_dtu_x_one_minus_dtv = (one_minus_dtu) * (one_minus_dtv);
int dtu_x_one_minus_dtv      = (dtu)      * (one_minus_dtv);
int dtu_x_dtv          = (dtu) * (dtv);
int one_minus_dtu_x_dtv      = (one_minus_dtu) * (dtv);

// 现在可以对纹理进行采样了
int r_texel = one_minus_dtu_x_one_minus_dtv * r_texel00 +
      dtu_x_one_minus_dtv      * r_texel10 +
      dtu_x_dtv          * r_texel11 +
      one_minus_dtu_x_dtv      * r_texel01;

int g_texel = one_minus_dtu_x_one_minus_dtv * g_texel00 +
      dtu_x_one_minus_dtv      * g_texel10 +
      dtu_x_dtv          * g_texel11 +
      one_minus_dtu_x_dtv      * g_texel01;

int b_texel = one_minus_dtu_x_one_minus_dtv * b_texel00 +
      dtu_x_one_minus_dtv      * b_texel10 +
      dtu_x_dtv          * b_texel11 +
      one_minus_dtu_x_dtv      * b_texel01;

// 绘制像素
screen_ptr[xi] = ((r_texel >> 16) << 11) +
         ((g_texel >> 16) << 5) +
         (b_texel >> 16);

// 更新 z 缓存
z_ptr[xi] = zi;
} // end if
```

上述代码计算纹理坐标的小数部分，然后在 RGB 空间计算最终的像素值，并将其绘制到屏幕上。这种算法的问题不在于其固有的复杂性，而在于计算量太大，因此难以优化。实际上，这是一种图像处理算法，不可能实时地完成。如果读者查看代码，将发现进行了大量的优化，如预先计算共用的因子，同时使用了 24.8 定点格式。之所以使用这种格式，是因为计算每个双线性插值项时，需要将三个数相乘，如果其中的两个为 16.16 格式，结果将溢出；为避免溢出，使用了 24.8 格式。双线性插值的效果非常好，这种演示程序看起来特别棒。

下面是另外两个支持双线性插值的纹理映射函数的原型，它们分别使用 1/z 缓存和不进行排序：

```
void Draw_Textured_Bilerp_TriangleINVZB_16(POLYF4DV2_PTR face, // 指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch,     // 每行多少字节(320、640 等)
            UCHAR *zbuffer,     // 指向 z 缓存的指针
            int zpitch);      // z 缓存中每行多少字节

void Draw_Textured_Bilerp_Triangle_16(POLYF4DV2_PTR face, // 指向多边形面的指针
            UCHAR *_dest_buffer, // 指向视频缓存的指针
            int mem_pitch);     // 每行多少字节(320、640 等)
```

DEMOII12_9.CPP|EXE 是一个使用双线性插值的演示程序，图 12.35 是其运行时的屏幕截图。用户可以按 B 键在点采样和双线性滤波纹理映射之间进行切换，还可以按 N 键来显示下一组物体。要编译该程序，需要 DEMOII12_9.CPP|H、T3DLIB1-10.CPP|H 和 DirectX .LIB 文件。

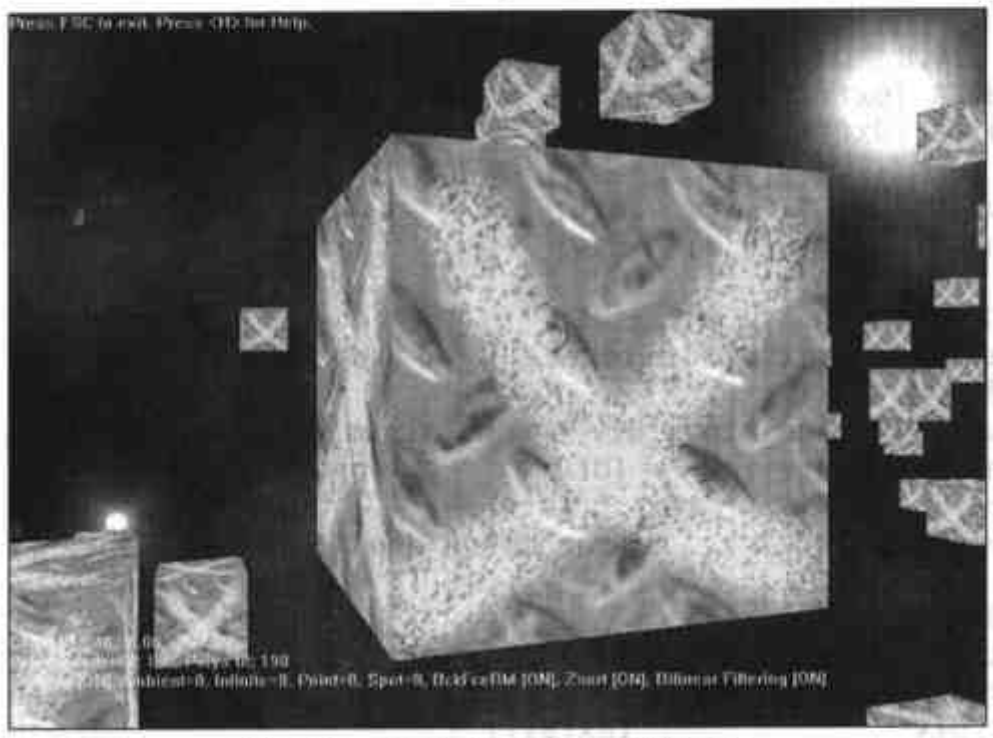


图 12.35　使用双线性插值可消除锯齿

12.7　Mipmapping 和三线性纹理滤波

Mipmapping 来自成语 multum in parvo，意思是“麻雀虽小，五脏俱全”。Mipmapping 用于解决走样的问题。将纹理映射到多边形上和对纹理进行采样（点采样、双线性滤波等）时，将出现一些与空间信息采样理论相关的怪异现象。进行纹理映射时，可能出现的问题之一是闪烁（sparkling）。当纹理图中的高频元素（即在纹理图中移动时，强度剧烈地变化）进入和离开视野以及纹理图被缩小时，将发生闪烁。基本上，这实际上是高频信息与采样频率时而同相时而异相的表现。

纹理映射的第二个主要问题是低频走样，导致波纹图案。与高频图像生成低频图像时将出现这种现象。图 12.36 是一个使用点采样得到的棋盘图像，注意到其中的波形图案吗？现在来看图 12.37，它描述的是同一个场景，但使用了 Mipmapping 和纹理滤波。波纹现象不那么严重，图像清晰得多。

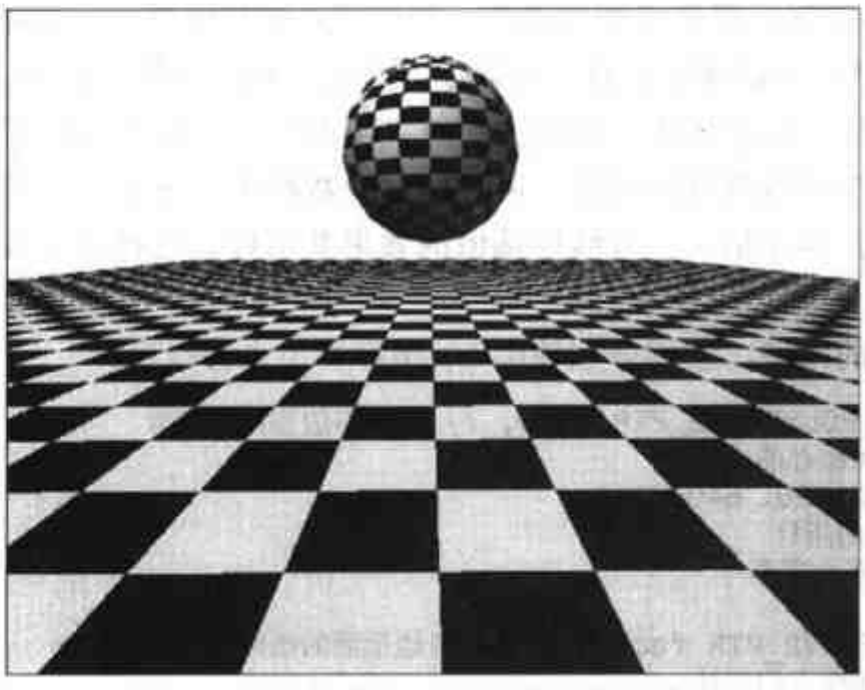


图 12.36　以点采样方式绘制的图像很糟糕

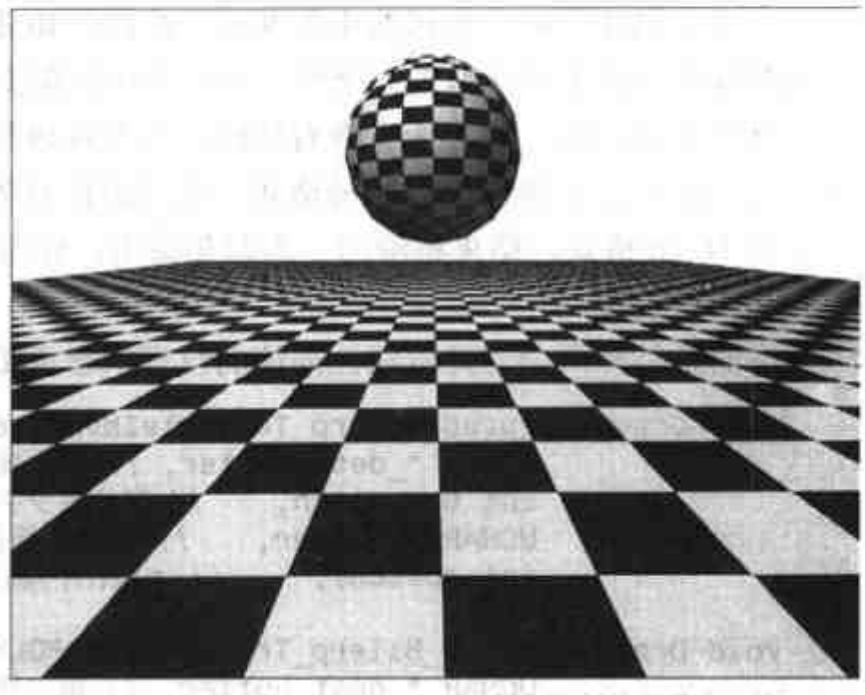


图 12.37　使用 Mipmapping 和滤波绘制的图像很不错

注意：将两个相同的重复图样（由直线、圆或点阵组成）拼贴起来时，如果它们没有完全对齐，将形成波纹图案（moire pattern）。波纹图案不是图像本身固有的，而是一种错觉。这种图案通常出现在对比度较高或频率较高的区域；将两幅图像拼贴起来时，由于浅色区域和深色区域的相互干扰，也会出现这种图案。

为实现 Mipmapping，需要创建由纹理组成的 mip 链，其中每个纹理的大小都为前一个纹理的 1/4（沿每条轴缩小一半），最后一个纹理的大小为 1×1。另外，这些 mip 纹理都是使用滤波器（平均滤波器、箱形滤波器、高斯滤波器等）生成的。渲染多边形时，根据多边形离视点的距离或多边形被投影后的面积，选择使用合适的 mip 纹理。这样，将最大限度地减少闪烁和低频走样/波形图案。接下来介绍具体的细节。

12.7.1　傅立叶分析和走样简介

这里只简要地介绍傅立叶分析为何重要、它有何功能、能提供什么样的信息以及我们为何关心它。

傅立叶分析将时间域（讨论图形学时，为空间域）中的信号分解为正弦分量。也就是说，在很大程度上说，任何满足 Dirchlet 条件的周期信号都可以分解为一系列频率、振幅和相位各不相同的正弦波。换句话说，任何信号都是由一系列正弦波组成的。因此，任何信号都可以用傅立叶级数来表示，而傅立叶级数是一系列正弦波的和。

通过傅立叶变换，可以提取信号的频率信息，进而计算出信号的组成部分。例如，假设有一个频率为 f（周期 T 为 $1/f$）的正弦波，如图 12.38a 所示。

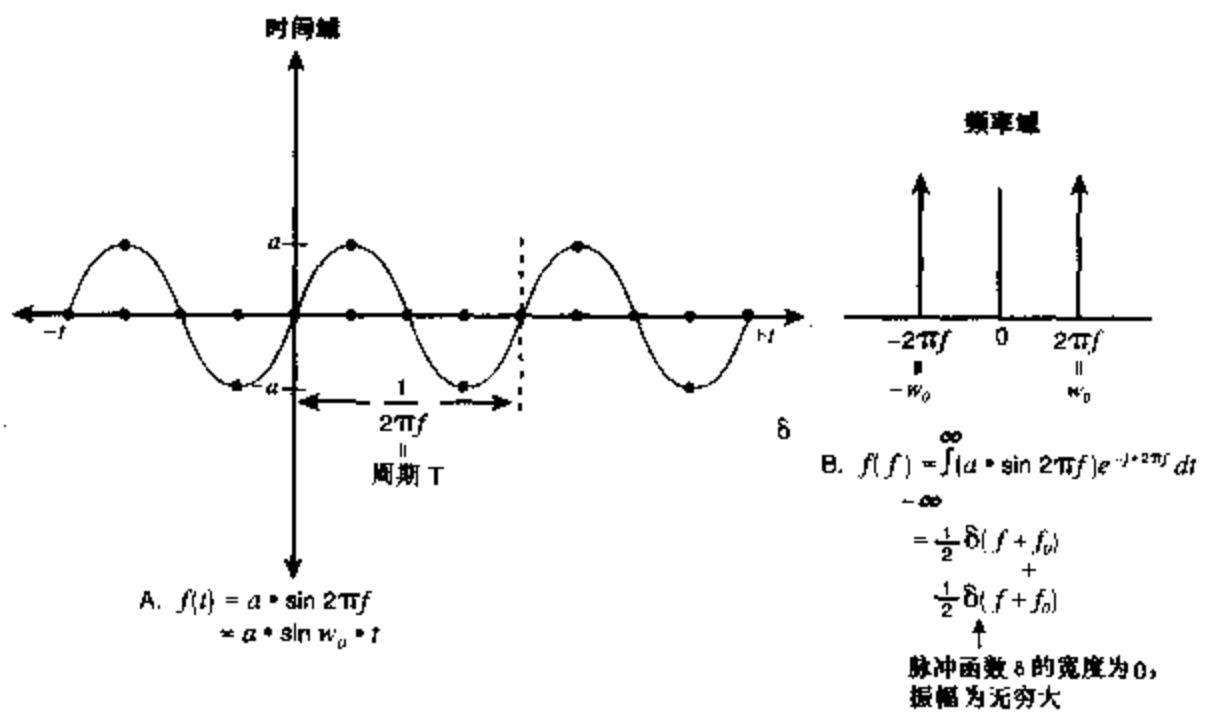


图 12.38　普通的正弦波及其傅立叶变换

这个信号位于时间域中，即其时间（t）函数如下：

```
f(t) = a*sin(2*π*f*t)
```

其中 a 为振幅，t 为时间，f 为频率。如果对这个简单的正弦函数进行傅立叶变换，结果将为单个值，它位于频率轴上的 f 处，振幅为 a。因此，用图形来表示傅立叶变换结果时，不使用时间轴，而是使用频率轴。图 12.38b 是对上述正弦函数进行傅立叶变换的结果。如果对下述函数进行傅立叶变换：

```
f(t) = 10*sin(100*t) + 20*sin(200*t)
```

将得到两个频率脉冲，它们分别位于频率域中的(100，200)和(–100，–200)处。任何信号都可分解为一系列的正弦波，使用这些正弦波可以重建原来的信号。进行傅立叶变换，将信号从时间域转换到频率域中的运算如下：

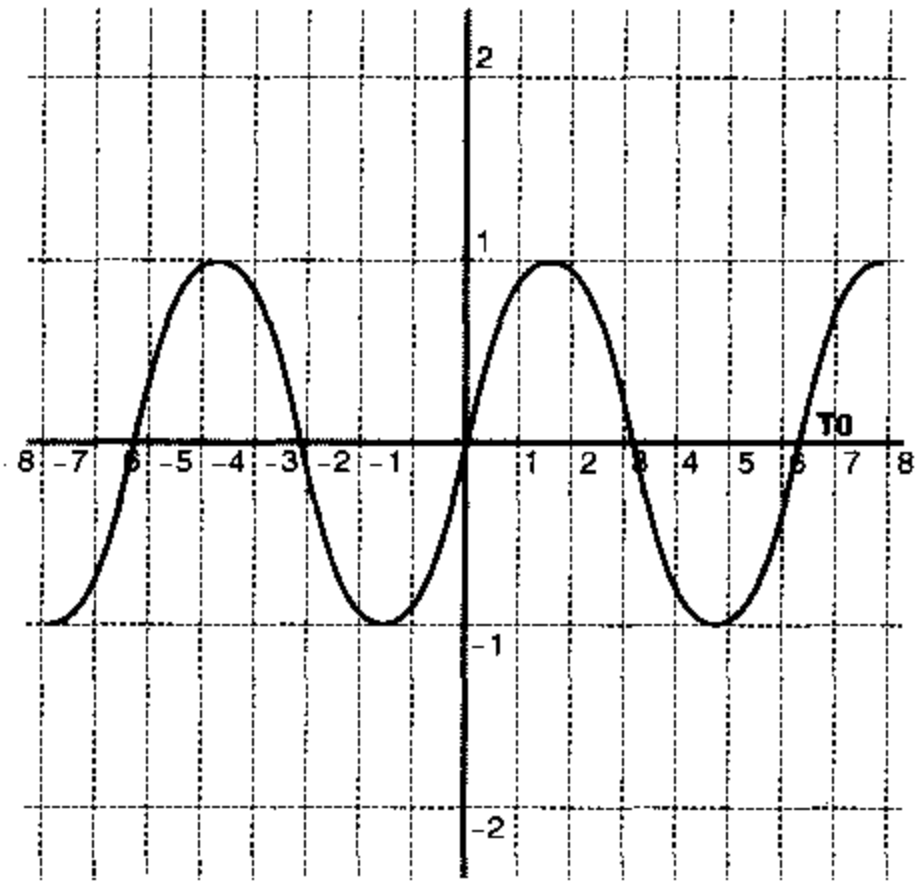
$$F\{\ f(t)\ \} = \int_{-\infty}^{+\infty} f(t) * e^{-j*2*\pi*t} dt$$

要计算傅立叶变换，只需对 f(t)进行积分，其中 j 是一个虚数，等于–1 的平方根。执行 Mipmapping 时，几乎无需知道如何进行傅立叶变换。傅立叶变换只是提供了一种提取信号中频率分量的数学方法，我们本来需要使用这种工具；但已经有人为我们完成了这些工作。下面来讨论结果。

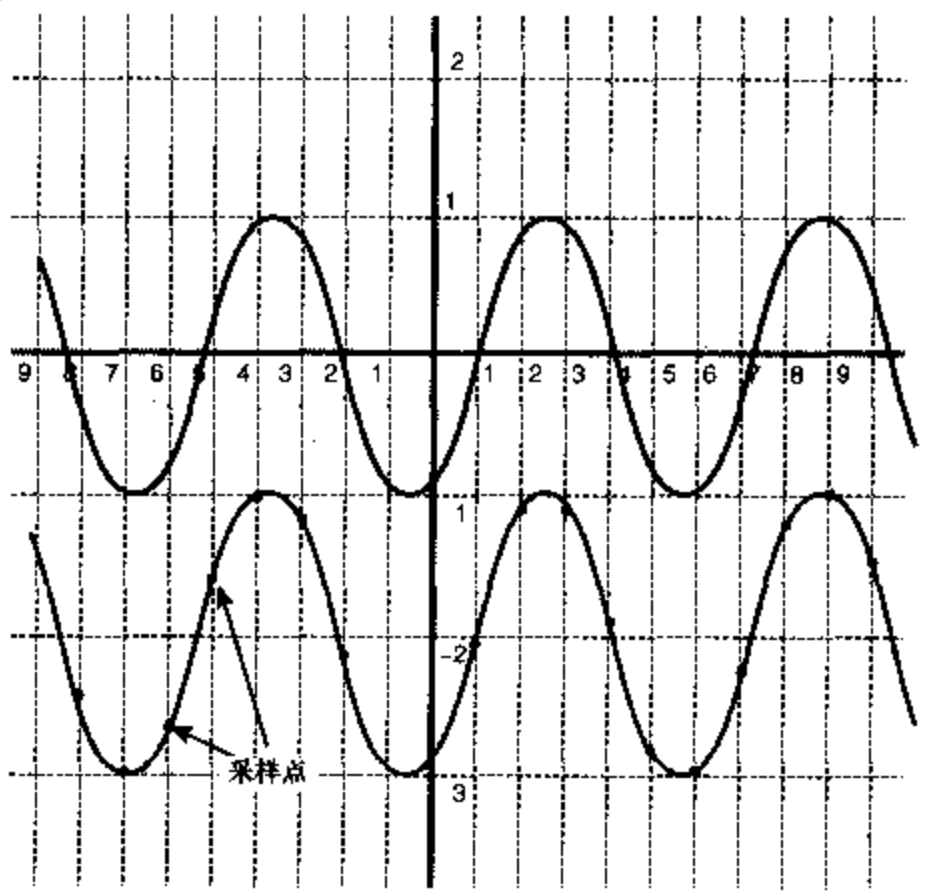
请看图 12.39。图 12.39a 是要对其进行采样的信号（可能是音频、图像或其他东西）。该信号的频率为 *f0*，周期 T0 = 1/*f0*。图 12.39b 显示了原来的信号以及以大于 2**f0* 的频率采样得到的信号，正如读者看到的，重建的信号与原来的信号几乎相同。将采样频率降低到 2**f0* 时，结果如图 12.39c 所示。非常凑巧的是，采样点的值全部为 0，因此重建的信号为水平线。

最后，来看一下图 12.39d，其中的采样频率小于 2**f0*，因此重建的信号为原始信号的低频走样版本。使用点采样时，当图像缩小时发生的情况与此相同：最后的图像将发生低频走样。

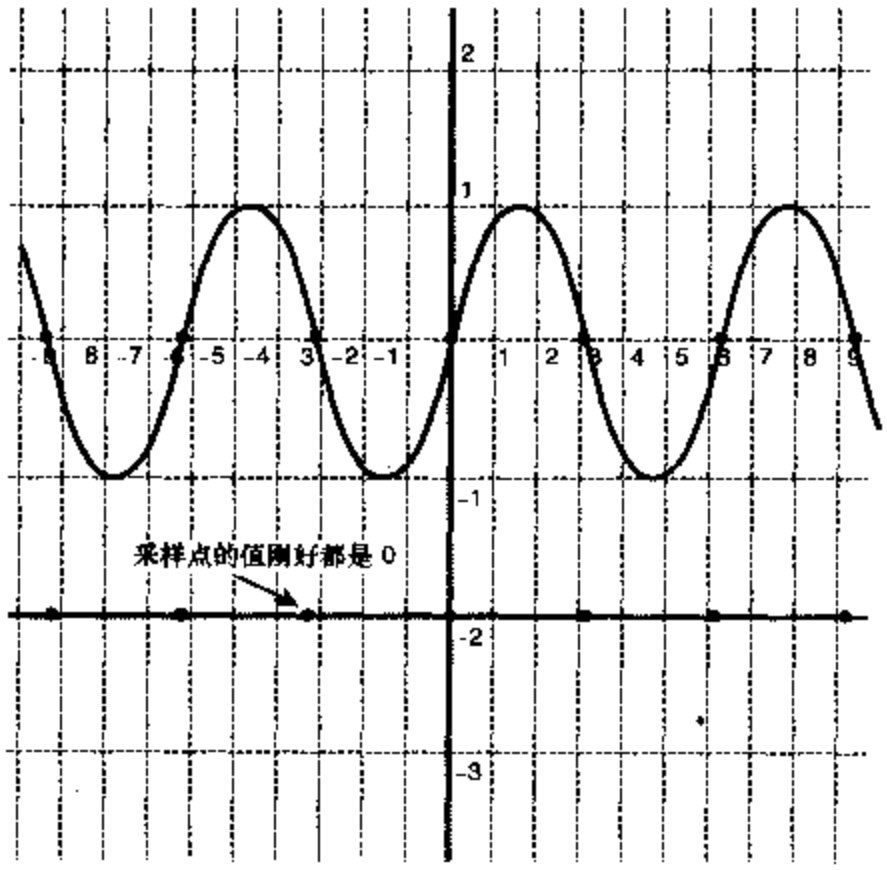
从很大程度上说，这种问题没有办法避免，不管采样频率多高，都将发生走样。我们能做的只能是尽可能地降低走样程度。为此，采取的技巧是删除高频信息，当纹理的缩小比例非常高时，这种信息会导致走样。这是 Mipmapping 的基本原理：根据原来的纹理创建一系包含的高频信息越来越少的纹理，其中每个纹理的大小都为前一个纹理的 1/4。



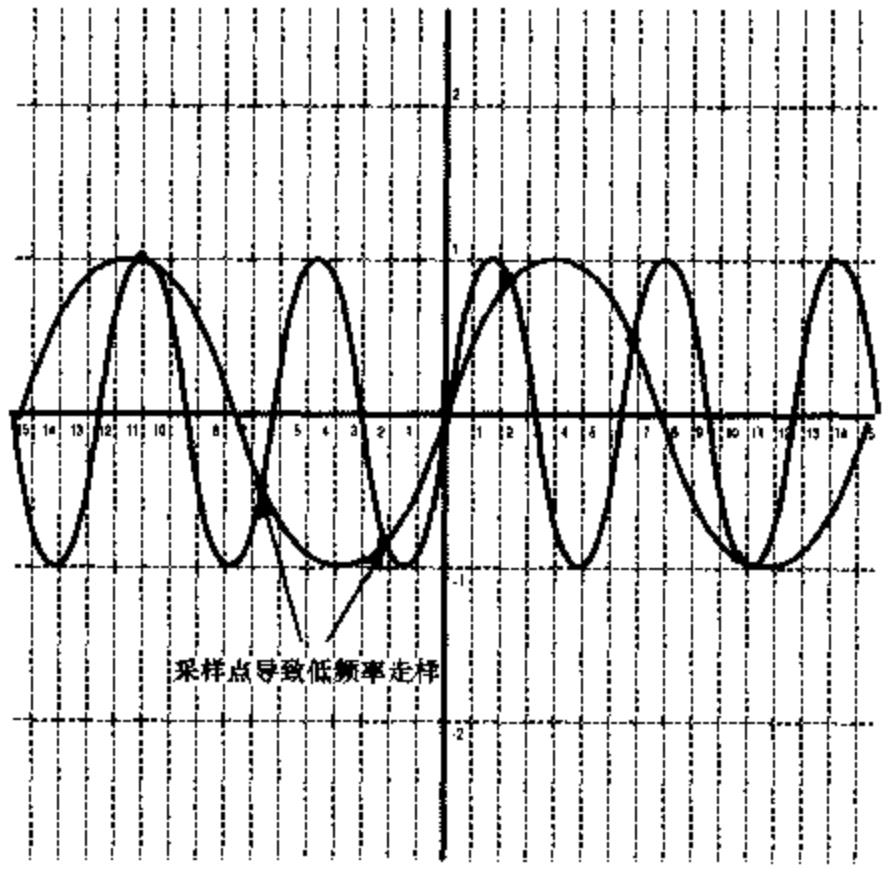
A. 要对其进行采样的信号



B. 采样频率大于 2*f0



C. 采样频率等于 2*f0



D. 采样频率小于 2*f0

图 12.39　对信号进行采样时发生的情况

12.7.2　创建 Mip 纹理链

我们要做的是，根据原始纹理 T0 创建一系列的纹理（通常使用平均滤波）：T1、T2...T*n*，其中每个纹理的大小都是前一个纹理的 1/4，即长度和宽度减半，如图 12.40 所示。

要根据前一个 mip 纹理计算当前纹理中纹素的值，可以使用平均滤波器，即在 RGB 空间中，计算纹素 (x, y)、$(x+1, y)$、$(x+1, y+1)$ 和 $(x, y+1)$ 的平均值，然后将结果写入到当前纹理中，如图 12.41 所示。

创建 Mip 纹理链时需要遵守一些规则。首先，所有纹理都必须是方形的，且边长为 2 的幂。这样，可以在两个轴上按相同的比例缩小，Mip 纹理链末尾的最后一个纹理总是 1×1 的。另一个约定是，原始纹理为 mip 等级 0，然后依次为 mip 等级 1、2、3、4...*n*。例如，如果原始纹理的大小为 256×256，则 mip 链中各个纹理的大小如表 12.4 所示。

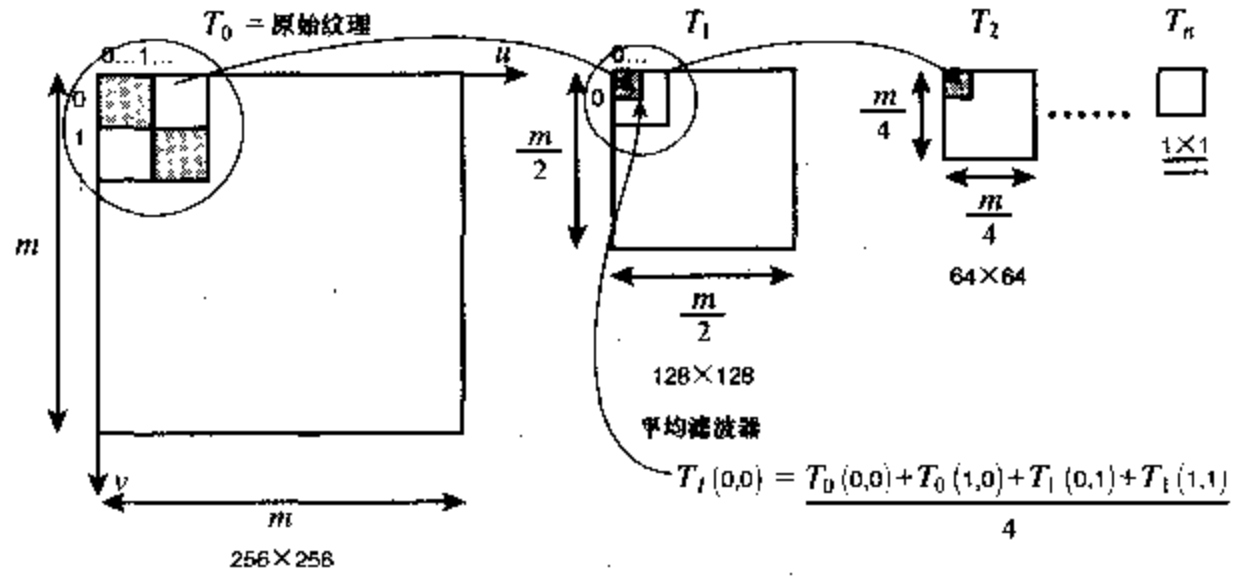


图 12.40　根据原始纹理创建 Mip 纹理链

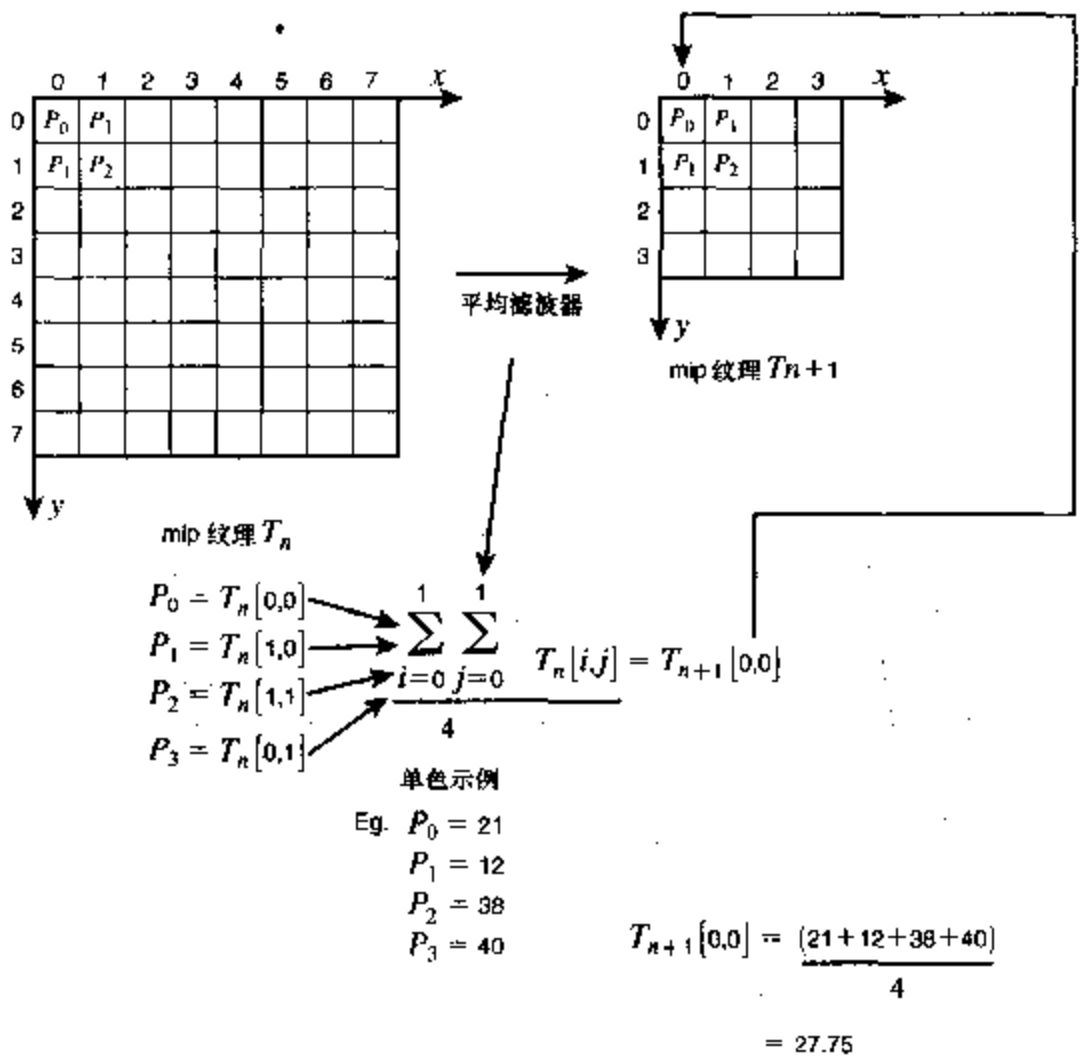


图 12.41　用于创建 mip 纹理的平均滤波器

表 12.4　　原始纹理为 256 x 256 时，各个 Mip 纹理的大小

Mip 纹理	纹理大小	Mip 纹理	纹理大小
T0	256×256	T5	8×8
T1	128×128	T6	4×4
T2	64×64	T7	2×2
T3	32×32	T8	1×1
T4	16×16		

除原始纹理外，mip 等级数为 $\log_2{}^{\text{T0 的边长}}$。

在这个例子中，为 $\log_2{}^{256} = 8$。要计算总纹理数，只需将上述结果加 1。因此计算总纹理数的公式为：$log_2{}^{\text{T0 的边长}}+1$。在这个例子中，总纹理数为 9。

这里纹理要占用多少内存呢？图 12.42 是实际的 mip 纹理，其中每个纹理的大小都是前一个纹理的 1/4。由于每次都将纹理缩小到原来的 1/4，因此占用的内存很少。下面计算在前一个例子中，纹理需要占用多少内存。

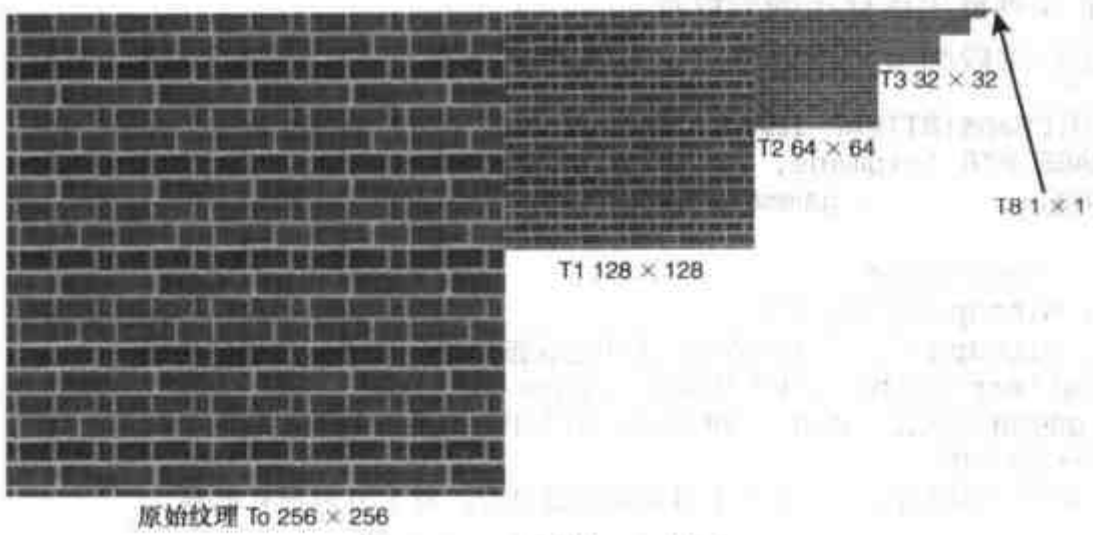


图 12.42　实际的 mip 纹理

初始纹理的大小为 256×256，需要占用 256^2 个字。Mip 纹理链需要的内存量如下：

$$x = 256^2 + 128^2 + 64^2 + 32^2 + 16^2 + 8^2 + 4^2 + 2^2 + 1^2$$

要计算 x 为原始纹理所需内存量的多少倍，将其除以 256^2：

$$\frac{x}{256^2} = \frac{256^2 + 128^2 + 64^2 + 32^2 + 16^2 + 8^2 + 4^2 + 2^2 + 1^2}{256^2}$$

$$= 1.333$$

换句话说，加上 mip 纹理时，每个纹理需要的内存量只增加 33%，代价很小，收益却很高。

知道如何创建 mip 纹理以及它们占用的内存量后，需要拿出一种将其加入到引擎中的方法。作者苦思冥想，考虑过再次修改物体/多边形数据结构，最后终于柳暗花明：只需将纹理指针指向 mip 纹理链而不是单个纹理即可。然而，在每个多边形中设置一个 Mipmapping 标记，并在渲染多边形时，将纹理指针视为执行位图数组的指针，而不是单个位图的指针。这样，便可以使用原来的数据结构。请看图 12.43，以理解这里讨论的方法。

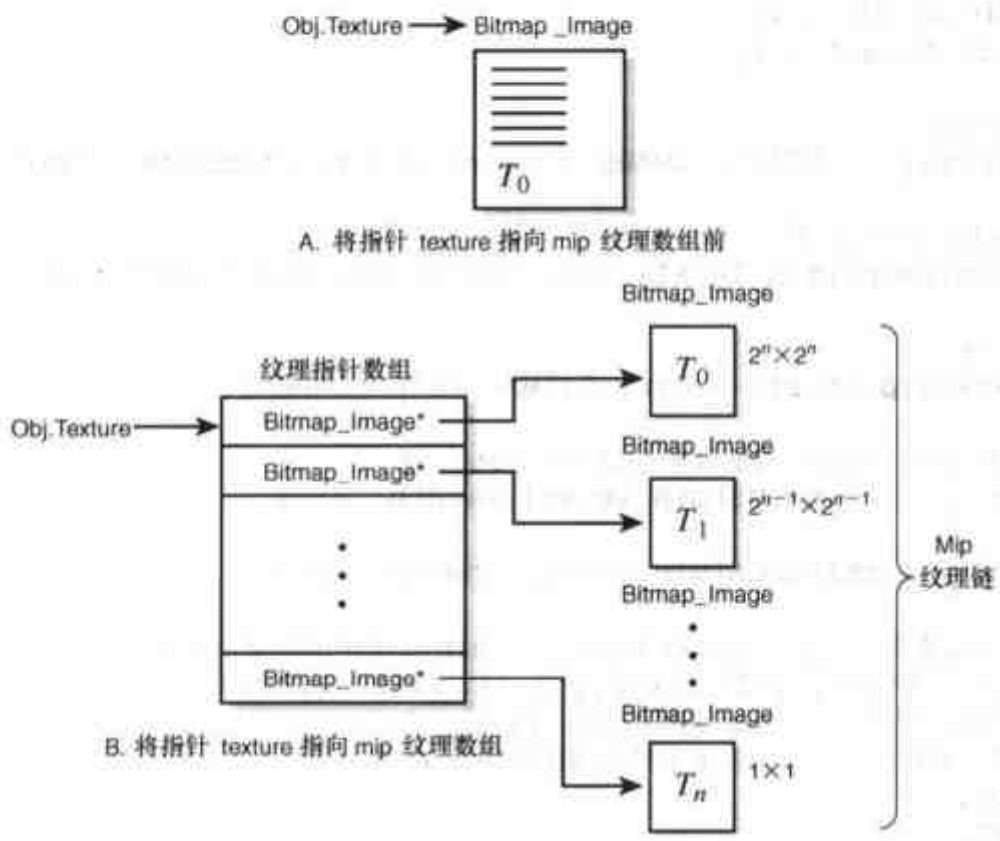


图 12.43　支持 mip 纹理链的纹理指针数组

需要对纹理进行 Mipmapping 时，原来的 texture 指针指向 0 级别 mip 纹理。据此分两步创建 mip 纹理链：首先，分配一个位图指针数组，并将原来的 texture 指针指向该数组；然后为每个 mip 纹理分配内存、生成 mip 纹理，并将位图指针数组中的每个元素指向相应 mip 纹理的位图。看起来令人迷惑，其实并非如此。

这种方案存在的惟一问题是，我们强行将物体/多边形的位图图像指针指向了一个位图指针数组。因此，必须非常小心，避免不知道纹理已被 mipmap 化的函数将 0 级纹理视为普通纹理；而知道纹理已被 mipmap 化的函数必须从 mip 纹理链中选择正确的纹理。

这涉及到一些运行阶段细节，稍后将讨论。现在先来看一下生成 mip 纹理的函数：

```
int Generate_Mipmaps(BITMAP_IMAGE_PTR source,    // 原始纹理
    BITMAP_IMAGE_PTR *mipmaps, // 指向 mip 纹理数组的指针
    float gamma)          // gamma 修正因子
{
// 这个函数创建一个 mip 纹理链
// 调用该函数时，mipmap 指向原始纹理
// 该函数退出时，mipmap 指向一个指针数组，其中包含指向各个 mip 级纹理的指针
// 另外，该函数返回 mip 等级数，如果发生错误，则返回-1
// 最后一个参数 gamma 用于提高 mip 纹理的亮度，因为平均滤波器会降低亮度
// 1.01 通常是不错的选择
// 该参数大于 1.0 时，将提高亮度；小于 1.0 时将降低亮度；为 1.0 时没有影响

BITMAP_IMAGE_PTR *tmipmaps; // 局部变量，指向指针数组的指针

// 第一步：计算 mip 等级数
int num_mip_levels = logbase2ofx[source->width] + 1;

// 为指针数组分配内存
tmipmaps = (BITMAP_IMAGE_PTR *)malloc(num_mip_levels *
      sizeof(BITMAP_IMAGE_PTR) );

// 将元素 0 指向原始纹理
tmipmaps[0] = source;

// 设置宽度和高度（它们相同）
int mip_width  = source->width;
int mip_height = source->height;

// 使用平均滤波器生成各个 mip 纹理
for (int mip_level = 1; mip_level <  num_mip_levels; mip_level++)
  {
  // 计算下一个 mip 纹理的大小
  mip_width  = mip_width  / 2;
  mip_height = mip_height / 2;

  // 为位图对象分配内存
  tmipmaps[mip_level] = (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE) );

  // 创建用于存储 mip 纹理的位图
  Create_Bitmap(tmipmaps[mip_level],0,0, mip_width, mip_height, 16);

  // 让位图可用于渲染
  SET_BIT(tmipmaps[mip_level]->attr, BITMAP_ATTR_LOADED);

  // 遍历前一个 mip 纹理，使用平均滤波器创建当前 mip 纹理
  for (int x = 0; x < tmipmaps[mip_level]->width; x++)
    {
  for (int y = 0; y < tmipmaps[mip_level]->height; y++)
    {
    // 需要计算 4 个纹素的平均值，这些纹素在前一个 mip 纹理中的位置如下：
    // (x*2, y*2)、(x*2+1, y*2)、(x*2,y*2+1)、(x*2+1,y*2+1)
    // 然后将计算结果写入到当前 mip 纹理的(x, y)处
    float r0, g0, b0,       // 4 个样本纹素的 R、G、B 分量
       r1, g1, b1,
       r2, g2, b2,
       r3, g3, b3;

    int r_avg, g_avg, b_avg; // 用于存储平均值

    USHORT *src_buffer  = (USHORT *)tmipmaps[mip_level-1]->buffer,
        *dest_buffer = (USHORT *)tmipmaps[mip_level]->buffer;
```

```
        // 提取每个纹素的 R、G、B 值
        _RGB565FROM16BIT( src_buffer[(x*2+0) + (y*2+0)*mip_width*2] ,
                 &r0, &g0, &b0);
        _RGB565FROM16BIT( src_buffer[(x*2+1) + (y*2+0)*mip_width*2] ,
                 &r1, &g1, &b1);
        _RGB565FROM16BIT( src_buffer[(x*2+0) + (y*2+1)*mip_width*2] ,
                 &r2, &g2, &b2);
        _RGB565FROM16BIT( src_buffer[(x*2+1) + (y*2+1)*mip_width*2] ,
                 &r3, &g3, &b3);

        // 计算平均值，并考虑 gamma 参数
        r_avg = (int)(0.5f + gamma*(r0+r1+r2+r3)/4);
        g_avg = (int)(0.5f + gamma*(g0+g1+g2+g3)/4);
        b_avg = (int)(0.5f + gamma*(b0+b1+b2+b3)/4);

        // 根据 5.6.5 格式，对 R、G、B 值进行截取
        if (r_avg > 31) r_avg = 31;
        if (g_avg > 63) g_avg = 63;
        if (b_avg > 31) b_avg = 31;

        // 写入数据
        dest_buffer[x + y*mip_width] = _RGB16BIT565(r_avg,g_avg,b_avg);

        } // end for y

      } // end for x

    } // end for mip_level

    // 让 mipmaps 指向指针数组
    *mipmaps = (BITMAP_IMAGE_PTR)tmipmaps;

    // 成功返回
    return(num_mip_levels);

} // end Generate_Mipmaps
```

该函数接受三个参数，它们有些棘手。第一个参数是位图指针 source，可以是指向任何有效 BITMAP_IMAGE 对象的指针，切记它是一个指针。第二个参数要复杂些：

```
BITMAP_IMAGE_PTR *mipmaps;
```

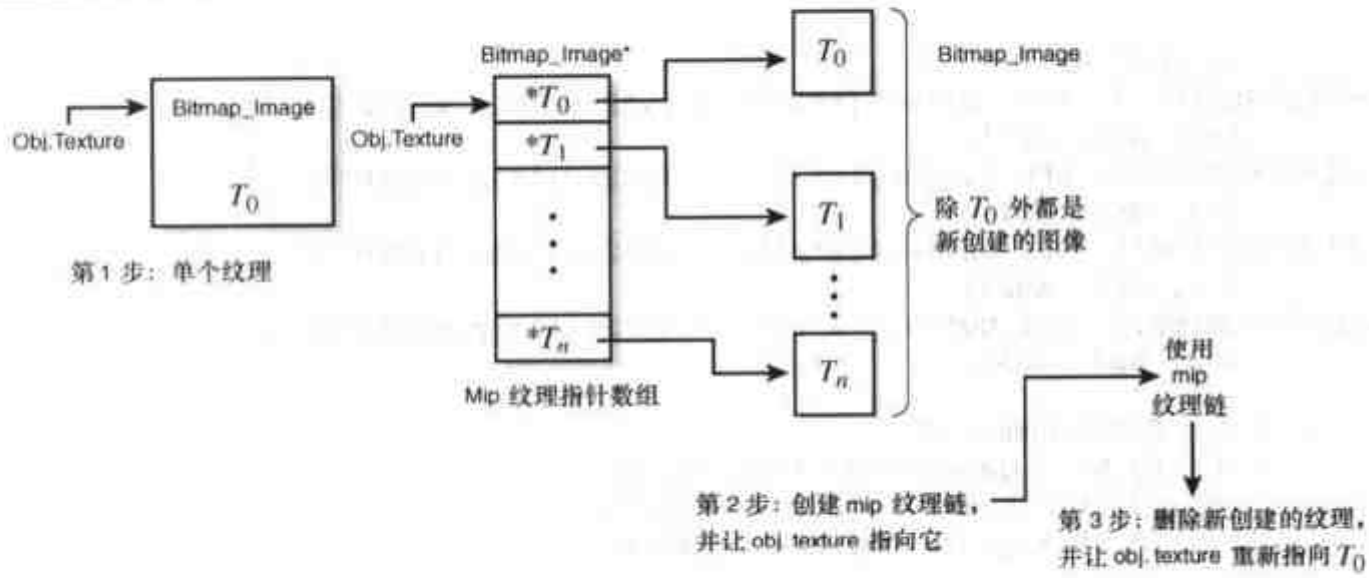
这是一个指向 BITMAP_IMAGE 指针的指针。假设物体（或多边形）有一个 texture 指针，它指向一个 BITMAP_IMAGE。现在的问题是，当 mip 纹理函数生成所有的 mip 纹理时，我们要将该指针指向 mip 纹理链本身，为此，需要知道该指针的地址，以便能够修改它，因此需要一个** BITMAP_IMAGE。这就是需要一个指向指针的指针作为参数的原因。我们将让该参数指向 mip 纹理指针数组。

这个函数提供了这样的灵活性：可以将同一个对象作为参数 source 和 mipmap。对于 source 参数，将其设置为指向该对象的指针；对于参数 mipmaps，需要将其设置为指向该对象的指针的地址，这样函数才能对其进行修改。假设您这样调用该函数：

```
Generate_Mipmaps(obj->texture, (BITMAP_IMAGE_PTR *)&obj->texture,1.01);
```

其中 obj 的类型为 OBJECT4DV2_PTR，它有一个 texture 字段，该字段是一个指向 BITMAP_IMAGE 的指针（BITMAP_IMAGE_PTR）。仔细查看上述调用可以发现：我们将该指针作为第一个参数，同时将其地址作为第二个参数，因此对第二个参数进行修改时，将修改 obj->texture 指向的实际值，从而丢失它指向的原始数据。但事实上，并不会丢失任何数据，我们将把 obj->texture 指向的原始位图用作 mip 纹理链中的第一个纹理。在删除 mip 纹理链时，我们重新将该指针指向原始位图。图 12.44 说明了整个处理过程。

回到 mip 纹理生成函数——它创建 mip 纹理链：为纹理指针数组分配内存；然后计算下一个 mip 等级的大小，为该 mip 等级分配内存，在 RGB 空间使用平均滤波器来创建该 mip 等级。这一过程不断重复，直到 mip 等级的大小为 1×1 为止，然后函数结束。该函数的最后一个参数（默认值为 1.01）用于设置 gamma



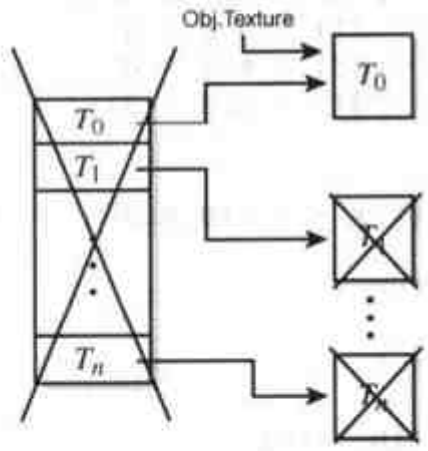


图 12.44　让 obj.texture 指向 mip 纹理链和重新指向 T_0

值。使用平均过滤器来不断缩小图像时，其亮度会逐渐降低；因此通常使用 gamma 因子来提高每个 mip 等级的亮度，确保所有 mip 等级的亮度一致。为说明这一点，作者编写了一个演示程序，名为 DEMOII12_10.CPP|H。用户可以选择不同的纹理图，该程序将动态地创建 mip 纹理，并显示它们，如图 12.45 所示。另外，用户还可以修改 gamma 值，以查看其影响。该程序的控制方式如下：

- 右箭头键——选择下一个纹理；
- 左箭头键——选择前一个纹理；
- 上箭头键——增大 gamma 值；
- 下箭头键——减小 gamma 值。

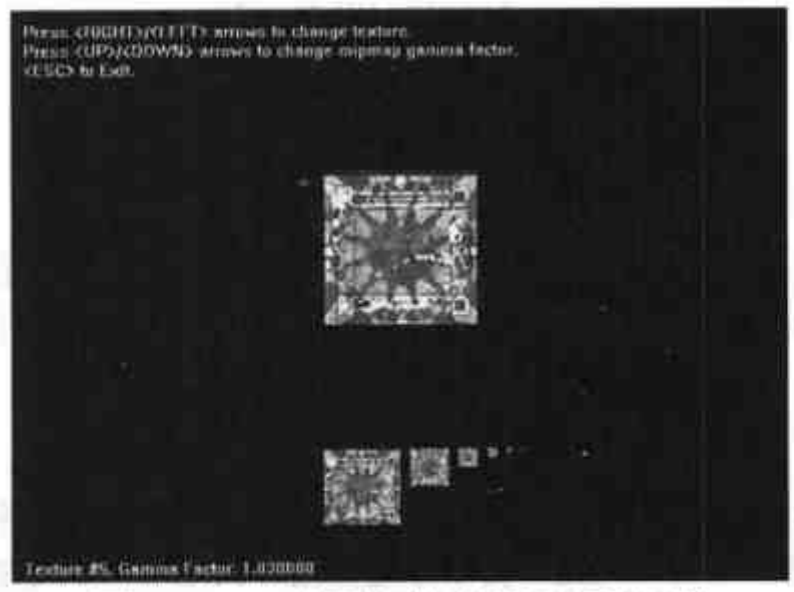


图 12.45　mip 纹理实时生成程序的屏幕截图

提示：每个 mip 纹理都是根据前一个纹理，使用平均滤波器计算得到的，大小从 m × m 变为 m/2 × m/2。这将逐渐降低图像的整体强度。在平均滤波处理期间，gamma 因子用作放大因子来提高纹理的亮度，以避免这种问题。

要编译该程序，需要 DEMOII12_10.CPP|H、库文件 T3DLIB1-10.CPP|H 和 DirectX .LIB 文件。

回到前面对函数 Generate_Mipmaps() 的调用。该函数执行后，obj->texture 将指向一个 mip 纹理链，而不是单个位图纹理。因此，如果现在调用光栅化函数，将得到错误的结果。我们需要对使用 Mipmapping 的多边形进行标记，并在调用光栅化函数之前检查这种标记。为此，定义了两个标记，一个用于多边形，一个用于物体：

```
// 支持 mipmapping 的新属性
#define POLY4DV2_ATTR_MIPMAP      0x0400 // 多边形是否有 mip 纹理链

#define OBJECT4DV2_ATTR_MIPMAP    0x0008 // 物体是否有 mip 纹理链
```

加载物体时，如果要对其使用 Mipmapping，必须在其属性中设置该标记；对于多边形也需要这样做。然而，这项工作由哪个函数来完成呢？读者可能猜到了，我们需要另一个支持 Mipmapping 的物体加载函数。由于 Caligari .COB 加载函数用得最多，作者只在这个函数中加入了这种功能。下面是新的函数原型：

```
int Load_OBJECT4DV2_COB2(OBJECT4DV2_PTR obj, // 指向物体的指针
       char *filename,     // Caligari COB 文件的名称
       VECTOR4D_PTR scale,  // 初始缩放因子
       VECTOR4D_PTR pos,    // 初始位置
       VECTOR4D_PTR rot,    // 初始旋转角度
       int vertex_flags,    // 顶点顺序反转和变换标记
       int mipmap=0);    // mipmapping 启用标记
                  //1 表示生成 mip 纹理，0 表示不生成
```

除最后一个参数是新增的外，该函数与前一版本相同，这个参数是一个标记，指出是否要使用 mipmapping。如果该参数为 1，且被加载的物体带有纹理，将使用 mipmapping；如果该参数为 0，将不使用 mipmapping。该物体加载函数非常大，这里无法列出其全部代码，但新增的代码块只有两个。第一个代码块设置物体的 mip 属性：

```
// 需要判断是否使用 mipmapping
// 如果是，则需要创建 mip 纹理链
// 并相应地设置物体和多边形的属性
if (mipmap==1)
  {
  // 设置物体的 mip 属性
  SET_BIT(obj->attr, OBJECT4DV2_ATTR_MIPMAP);

  // 调用 mip 纹理链生成函数
  Generate_Mipmaps(obj->texture, (BITMAP_IMAGE_PTR *)&obj->texture);
  } // end if
```

上述代码在材质分析代码之前。如果多边形带纹理，在设置多边形属性时，将设置 mip 属性，如下所示：

```
if (materials[ poly_material[curr_poly] ].attr & MATV1_ATTR_SHADE_MODE_TEXTURE)
  {
  // 设置着色模式
  SET_BIT(obj->plist[curr_poly].attr, POLY4DV2_ATTR_SHADE_MODE_TEXTURE);

  // 将纹理应用于多边形
  obj->plist[curr_poly].texture = obj->texture;

  // 设置多边形的 mipmap 属性

  if (mipmap)
    SET_BIT(obj->plist[curr_poly].attr,POLY4DV2_ATTR_MIPMAP);

  // 设置纹理坐标属性
  SET_BIT(obj->vlist_local[ obj->plist[curr_poly].vert[0] ].attr,
      VERTEX4DTV1_ATTR_TEXTURE);
  SET_BIT(obj->vlist_local[ obj->plist[curr_poly].vert[1] ].attr,
      VERTEX4DTV1_ATTR_TEXTURE);
  SET_BIT(obj->vlist_local[ obj->plist[curr_poly].vert[2] ].attr,
      VERTEX4DTV1_ATTR_TEXTURE);

} // end if
```

这就是新的物体加载函数中新增的全部内容：创建 mip 纹理和设置 mip 属性。棘手的地方是，如何选择 mip 纹理并将其传递给光栅化函数，稍后将介绍。现在来看一下与 mip 纹理相关的最后一个函数——删除函数：

```
int Delete_Mipmaps(BITMAP_IMAGE_PTR *mipmaps, int leave_level_0)
{
// 这个函数删除数组 mipmaps 中每个指针指向的纹理
// 然后释放该数组占用的内存
// 如果参数 leave_level_0 为 1，则保留 0 级纹理
```

```
BITMAP_IMAGE_PTR *tmipmaps = (BITMAP_IMAGE_PTR *)*mipmaps;

// mipmaps 指针是否有效?
if (!tmipmaps)
  return(0);

// 第 1 步: 计算 mip 等级数
int num_mip_levels = logbase2ofx[tmipmaps[0]->width] + 1;

// 遍历并删除每个位图
for (int mip_level = 1; mip_level < num_mip_levels; mip_level++)
  {
  // 释放位图占用的内存
  Destroy_Bitmap(tmipmaps[mip_level]);

  // 释放位图对象本身
  free(tmipmaps[mip_level]);

  } // end for mip_level

// 根据参数 leave_level_0 的值决定是否保留 0 级纹理
if (leave_level_0 == 1)
  {
  // 需要一个指向 0 级纹理的临时指针
  BITMAP_IMAGE_PTR temp = tmipmaps[0];

  // 让 mipmaps 指向 0 级纹理
  *tmipmaps = temp;
  } // end if
else
  {
  // 删除 0 级纹理
  Destroy_Bitmap(tmipmaps[0]);

  // 释放位图对象本身
  free(tmipmaps[0]);

  } // end else

// 成功返回
return(1);

} // end Delete_Mipmaps
```

函数 Delete_Mipmaps() 接受两个参数：一个指向位图图像指针的指针以及一个指示保留还是删除 0 级 mip 纹理的标记。这里需要指针地址的原因与 mip 纹理生成函数中类似。我们需要修改指针指向的内容，换句话说，我们需要恢复原样，让指针指向一个 BITMAP_IMAGE，而不是一个纹理指针数组。例如，假设有一个名为 obj 的带纹理的物体，您调用新的物体加载函数来加载它，并将参数 mipmap 设置为 1。该函数将根据物体的纹理创建一个 mip 纹理链，但使用完物体后，需要将其恢复到原样，因此需要删除 mip 纹理链中的元素 1–*n*，并将指针 texture 重新指向纹理 T0。

这就是需要该指针的地址而不仅仅是该指针的原因。函数 Delete_Mipmaps() 的参数 leave_level_0 用于指定是否要删除纹理 0 级纹理。如果该参数为 1，将保留 0 级纹理，并让指针 texture 指向它；如果为 0，将删除一切：mip 纹理数组、所有纹理（包括 0 级纹理），并将 texture 指针设置为 NULL。如果要求完美，则在启用 mipmapping 的情况下加载物体后，在退出程序或代码块时，应使用该物体的 texture 指针作为参数调用函数 Delete_Mipmaps()。

12.7.3 选择 mip 纹理

在渲染时，选择 mip 纹理的方式有两种：基于像素和基于多边形。前者在渲染每个像素时都使用一种算法来决定选择哪个 mip 纹理。完成这项工作的方法有多种，但都过于复杂，无法在软件引擎中实时地完成。我们将始终采用基于多边形来选择 mip 纹理的方法。

要找到合适的算法来选择合适的 mip 等级，必须首先牢记为何创建 mip 等级：通过使用平均滤波器每次将纹理缩小到原来的 1/4，然后在被渲染的投影多边形变小时，使用包含的高频信息更少、尺寸更小的纹理图，从而最大限度地降低高频走样（表现为低频走样或波形图案）。

将大小为 256×256 的纹理映射到一个 10×10 像素的多边形上是不合理的，这将导致扭曲、闪烁等现象。在这种情况下，使用 8×8 的纹理正合适。选择 mip 等级的算法如下：将多边形投影到屏幕后的面积与 0 级 mip 纹理的面积进行比较，得到纹素数和像素数之比，我们希望这种比例尽可能接近 1:1。换句话说，我们不想将 256×256 的纹理用于 10×10 的多边形，而想使用 8×8 的纹理。同样，我们也不想将 8×8 的纹理用于 30×30 的多边形，在这种情况下，32×32 的纹理更合适。

1. 根据纹素数和像素数之比选择 Mip 等级

现在来看一下能否根据每个 mip 等级包含的信息量为前一个等级的 1/4，推导出一个严密的公式。如果纹素数和像素数之比大于 4^n，需要使用第 n 个 mip 等级。从数学上说，计算纹素数和像素数之比的公式如下：

$$\text{mip\_ratio} = \frac{\text{0级mip纹理的面积（单位为纹素）}}{\text{多边形被投影后的面积（单位为像素）}}$$

例如，假设 0 级 mip 纹理的大小为 256×256，要将纹理映射到其上的多边形为 24×90，则 mip_ratio 如下：

```
mip_ratio = 256*256 / 24*90 = 65536 / 2160 = 30.3
```

我们知道，如果 mip_ratio 大于 4^n，则需要使用第 n 个 mip 等级。因此，要找出应使用的 mip 等级，需要计算 $\log_4 30.3$。幸运的是，存在如下所示的对数变换公式：

$$\log_a x = \ln x / \ln a$$

其中 ln 表示自然对数，即以 e = 2.718 为底的对数。

将上述变换公式应用于 $\log_4 30.3$，结果如下：

```
log4 30.3 = ln 30.3 / ln 4 = 2.46
```

mip 等级为 2.46，这意味着应使用 2 级 mip 纹理。从表 12.4 可知，2 级 mip 纹理的大小为 64×64，即面积为 4096，而多边形面积为 2160，它们的比值小于 4，这表明对于大小为 24×90 的多边形来说，我们选择的 64×64 mip 纹理是最合适的。

为证明这种算法可行，再来看两个简单的例子。如果多边形的大小为 200×170，将选择 0 级 mip 纹理，因为其面积与多边形面积的比例小于 4：

```
mip_ratio = (256*256) / (200*170) = 1.927
```

这表明应该选择 0 级 mip 纹理（256×256 的纹理）。下面来看另一个极端的例子。假设多边形投影后的大小为 3×4。我们预期应选择 6 或 7 级 mip 纹理，它们的大小分别为 4×4 和 2×2。来看看具体情况：

```
mip_ratio = (256*256) / (3*4) = 5461.33

log4 5461.33     = 6.2
```

这表明应选择 6 级 mip 纹理。

这种算法存在的惟一问题是，需要计算多边形（三角形）的面积，用它来除纹理的面积，然后执行对数运算，才能得到应选择的 mip 等级。虽然只需要几条指令就能完成，但完全没有必要。实际上，还有一个容易得多的方法，在 99%的情况下，使用这种方法来选择 mip 等级的效果都不错，而且通过调整，还可以进一步提高准确度。这种方法根据离视点的距离以线性衰减的方式来选择 mip 等级，下面来介绍它。

2. 根据 z 值选择 mip 纹理

当多边形变小时，我们需要在 n 个不同的 mip 纹理之间做出选择。通常，多边形离视点越远，它将变

得越小。当然，一个离视点很小的多边形和一个离视点很远的大型多边形看起来可能一样大，但这毕竟是极端情况。关键之处在于，可以根据不那么严密的度量值（如距离）而不是纹素数和像素数之比来选择 mip 等级。

如图 12.46 所示，我们将设置一个最大 mip 距离，当距离达到或超过这个值时，选择使用最后一个 mip 等级。换句话说，假设最大 mip 距离被设置为 10 000 个单位，则对于（最大或平均）z 坐标大于 10 000 的任何多边形，都将选择使用最高的 mip 等级（大小为 1×1 的纹理）。对于位于 0～10 000 之间的多边形，将根据其 z 坐标与最大 mip 距离之间的比值来选择 mip 等级。

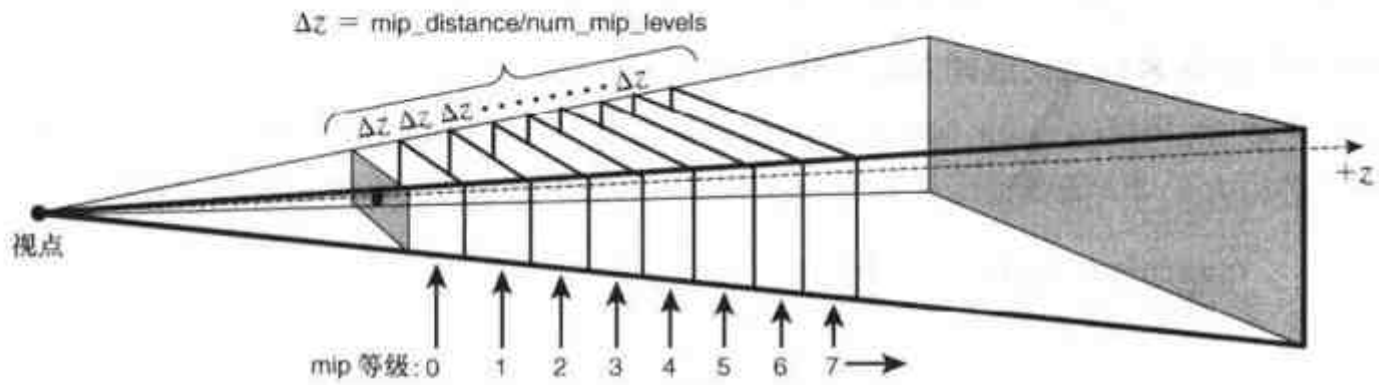


图 12.46　根据距离选择 mip 等级

下面是渲染场境函数 Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16() 中选择 mip 等级的代码：

```
// 检查多边形的 mipmap 属性
if (rc->rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_MIPMAP)
  {
  // 检查渲染场境的 mipmap 属性
  if (rc->attr & RENDER_ATTR_MIPMAP)
    {
    // 确定使用哪个 mip 纹理
    // 首先计算该多边形的 mip 纹理链的 mip 等级数
    int tmiplevels =
    logbase2ofx[((BITMAP_IMAGE_PTR *)
        (rc->rend_list->poly_ptrs[poly]->texture))[0]->width];

    // 根据最大 mip 距离和多边形的 z 坐标确定使用哪个 mip 等级
    // 这里使用多边形一个顶点的 z 坐标，读者可能想使用平均 z 值
    int miplevel = (tmiplevels *
           rc->rend_list->poly_ptrs[poly]->tvlist[0].z / rc->mip_dist);

    // 截取计算得到的 mip 等级
    if (miplevel > tmiplevels) miplevel = tmiplevels;

    // 根据 mip 等级选择相应的纹理
    face.texture = ((BITMAP_IMAGE_PTR *)
          (rc->rend_list->poly_ptrs[poly]->texture))[miplevel];

    // 根据 mip 等级，执行相应次数的"纹理坐标除以 2"运算
    for (int ts = 0; ts < miplevel; ts++)
     {
     face.tvlist[0].u0*=.5;
     face.tvlist[0].v0*=.5;

     face.tvlist[1].u0*=.5;
     face.tvlist[1].v0*=.5;

     face.tvlist[2].u0*=.5;
     face.tvlist[2].v0*=.5;
     } // end for

  } // end if mipmmaping enabled globally
  else // 渲染场景的 mipmap 属性没有被设置
  {
  // 用户没有请求使用 mipmaping，因此使用 0 级别纹理
  face.texture = ((BITMAP_IMAGE_PTR *)
```

```
            (rc->rend_list->poly_ptrs[poly]->texture))[0];

     } // end else

     } // end if
    else
     {
     face.texture = rc->rend_list->poly_ptrs[poly]->texture;
     } // end if
```

上述代码块位于调用光栅化函数之前，它判断是否需要使用 mipmapping。需要考虑的情形有多种。首先是多边形的 mipmap 属性没有被设置，在这种情况下，不能使用 mipmpping。其次是多边形的 mipmap 属性被设置，但没有使用渲染属性请求这种功能。最后是多边形的 mipmap 属性被设置，同时使用渲染属性请求了这种功能。在代码中必须考虑所有这些情形。下面依次介绍上述代码。首先，在代码块的开头，通过计算 0 级纹理宽度的以 2 为底的对数来确定 mip 等级数：

```
int tmiplevels =
   logbase2ofx[((BITMAP_IMAGE_PTR *)

         (rc->rend_list->poly_ptrs[poly]->texture))[0]->width];
```

这段难懂的代码将指针 texture 从纹理指针强制转换为指向纹理指针数组的指针，然后存取第一个纹理的宽度（第一个纹理总是 0 级纹理）。如果该纹理的大小为 128×128，则有 $\log_2 128 = 7$ 个 mip 等级。接下来的代码选择 mip 等级：

```
int miplevel = (tmiplevels * rc->rend_list->poly_ptrs[poly]->tvlist[0].z /

       rc->mip_dist);
```

上述代码根据渲染场境的参数 mip_dist 来缩小范围。该参数指出距离超过多少时，将使用最高 mip 等级的纹理（1×1 的纹理）。接下来，下述代码设置用于 face 的纹理：

```
// 根据 mip 等级选择相应的纹理
face.texture = ((BITMAP_IMAGE_PTR *)
       (rc->rend_list->poly_ptrs[poly]->texture))[miplevel];
```

同样，这里也对 texture 指针进行了强制转换，因为需要使用指向纹理指针数组的指针，而不是纹理指针。face.texure 是一个纹理指针，因此这里需要解决一些难题。

接下来的代码涉及一个非常重要的细节，我们还没有介绍过。假设纹理坐标(u，v)为(63，63)，而 0 级纹理的大小为 64×64，则(63，63)对应于纹理的右上角。然而，选择另一个 mip 等级后，其大小不再是 64×64，而可能是 8×8、2×2 等。因此，需要缩小纹理坐标，以便能够正确地进行映射，如图 12.47 所示。

执行这种功能的代码根据选择的 mip 等级与 0 级之间相差多少级，然后执行相应次数的“将纹理坐标除以 2（乘以 0.5）”运算，使纹理坐标与选择的纹理图匹配：

```
for (int ts = 0; ts < miplevel; ts++)
  {
  face.tvlist[0].u0*=.5;
  face.tvlist[0].v0*=.5;
  face.tvlist[1].u0*=.5;
  face.tvlist[1].v0*=.5;

  face.tvlist[2].u0*=.5;
  face.tvlist[2].v0*=.5;
  } // end for
```

显然，有更快的计算方法，例如使用其中包含 $(0.5)^{mip\_level}$ 的查找表，将纹理坐标与查找结果相乘，而不是重复执行多次乘法，但就现在而言，上述方法更容易理解。

注意：对纹理坐标、纹理等执行的操作都是针对局部变量 face 的，因此无需考虑恢复到原来状态的问题。

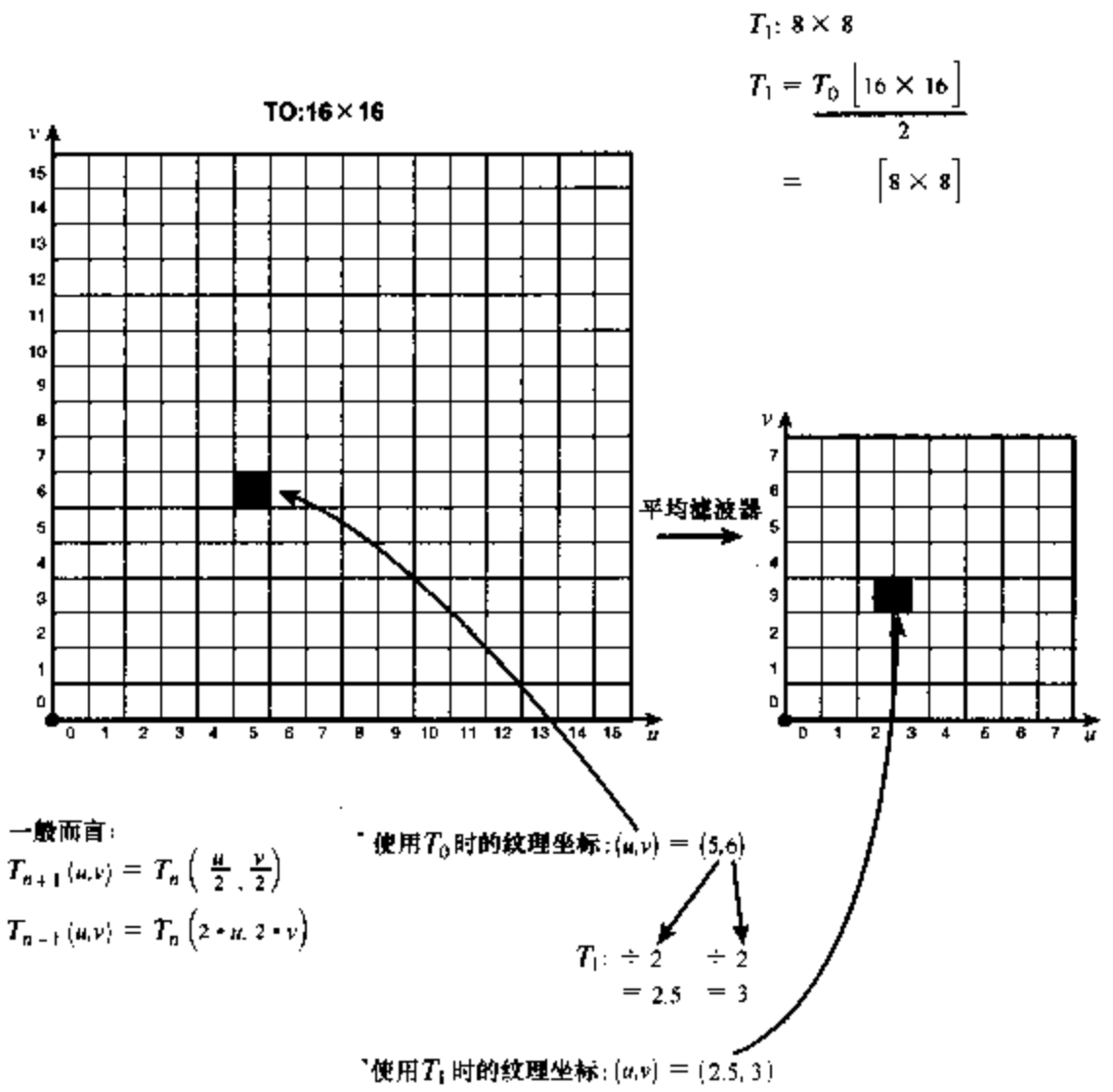


图 12.47　缩小纹理坐标，使之与选择的 mip 等级匹配

有关 mipmapping 就介绍到这里。读者可能更愿意使用根据面积比率选择 mip 等级的方法，而不是速度更快的第二种方法，使用哪种方法完全由读者决定。介绍演示程序之前，先来看一个调用物体加载函数的例子以及如何设置渲染场境。下面是一个调用物体加载函数的典型范例，其中启用了 mipmapping 功能：

```
// 加载物体
Load_OBJECT4DV2_COB2(&obj_scene, "cube_flat_textured_01.cob",
            &vscale, &vpos, &vrot, VERTEX_FLAGS_SWAP_YZ |
            VERTEX_FLAGS_TRANSFORM_LOCAL ,
             1); // 启用 mipmapping
```

接下来设置渲染场境：

```
// 设置渲染场境
rc.attr = RENDER_ATTR_ZBUFFER
    | RENDER_ATTR_MIPMAP
    | RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE;

  rc.video_buffer  = back_buffer;
  rc.lpitch      = back_lpitch;
  rc.mip_dist     = 1500;
  rc.zbuffer     = (UCHAR *)zbuffer.zbuffer;
  rc.zpitch      = WINDOW_WIDTH*4;
  rc.rend_list    = &rend_list;
  rc.texture_dist  = 0;
  rc.alpha_override = -1;

  // 渲染场景
  Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16(&rc);
```

DEMOII12_11.CPP|EXE 是一个演示 mipmapping 和双线性插值的程序，图 12.48 是该程序运行时的屏幕截图。用户可在 3D 世界中移动，并启用/禁用 mipmapping。

使用 mipmapping 后，纹理映射的质量将极大地提高。另外，该演示程序还支持双线性纹理映射，效果

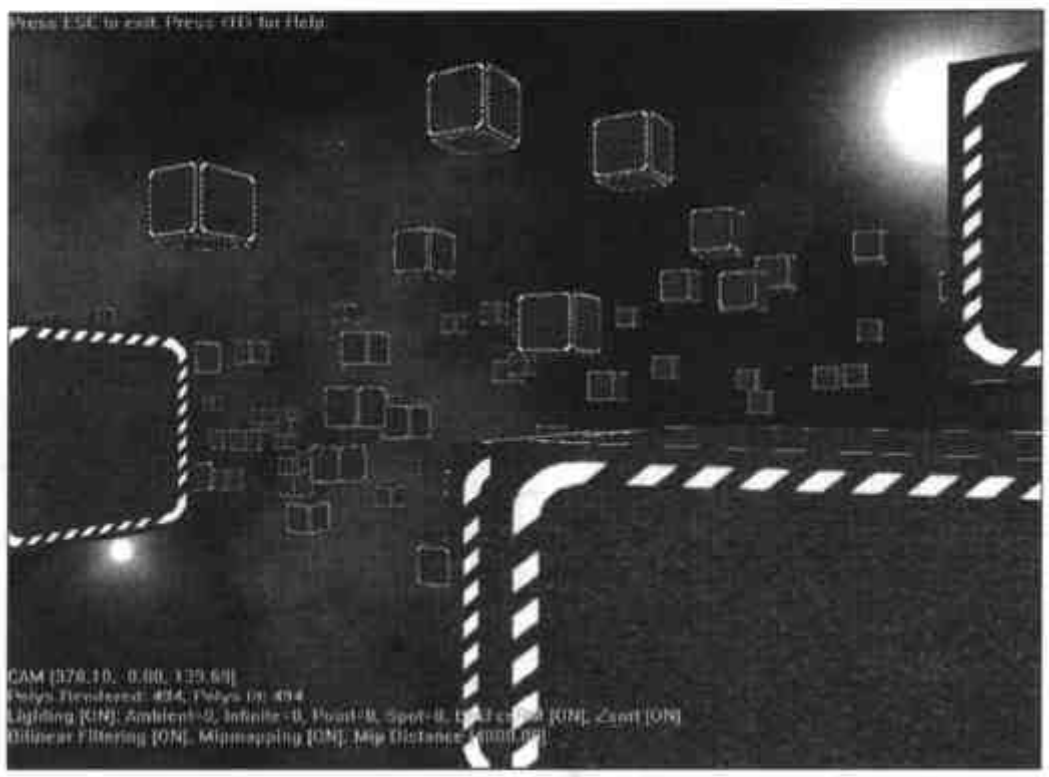


图 12.48　mipmapping 演示程序

非常好。程序的控制方法如下：

- 箭头键——移动；
- B——开/关双线性滤波；
- M——开/关 mipmapping；
- 1 和 2——加大/减小最大 mip 距离；
- N——选择要试验的下一个物体。

请尝试调整最大 mip 距离，以便在不同 mip 等级之间切换。这被称为 mip 跳跃（popping）；发生在多边形内时被称为 mip 分片（banding）。总之，根据距离来选择 mip 等级效果很好，与复杂的面积比率方法相比，其开销小得多。

提示：按 1 和 2 来修改最大 mip 距离。将其设置为 500～1000 时，更换 mip 等级的情况将频繁发生；对于游戏而言，3500 左右是不错的选择。

最后，要编译该程序，需要 DEMOII12_11.CPP|H、T3DLIB1-10.CPP|H 和 DirectX .LIB 文件。

12.7.4　三线性滤波

接下来简要地讨论一下三线性滤波。三线性滤波结合使用双线性插值和 mipmapping，是纹理滤波的极限。首先使用某种技术（很可能是面积技术，因为它的准确度较高）来选择渲染多边形或像素时使用的 mip 等级；最终的结果是类似 4.3 这样的数字，这表明应使用 mip 等级 4 和 5。也就是说，将根据 mip 纹理 4 和 5，使用线性插值来计算最终的像素值，公式如下：

```
mip_pixel = (1 - 0.3)* mip_level_pixel_4 + (.3)*mip_level_pixel_5
```

即不是使用一个 mip 纹理，而是根据计算得到的 mip 等级数确定应使用哪两个 mip 纹理，然后根据 mip 等级数的小数部分，对这两个 mip 纹理中相应的纹素进行线性插值，来计算最终的像素值。这就是三线性插值的由来：首先使用双线性插值选择纹理 4 中的纹素；然后使用双线性插值选择纹理 5 中的纹素；最后根据这两个纹素，通过线性插值计算得到最后的像素值。图 12.49 说明了整个处理过程。

当前，很多引擎/硬件实际上并没有这样做。有些引擎只是使用点采样从纹理 4 和 5 中提取纹素，然后通过线性插值计算出最终的像素值；而其他一些引擎执行上述所有计算。我们本可以立刻实现这种功能，但其速度太慢了。作为练习，读者应该能够实现它。问题在于，最后的插值必须在光栅化函数中进行，这正是导致困境的因素。

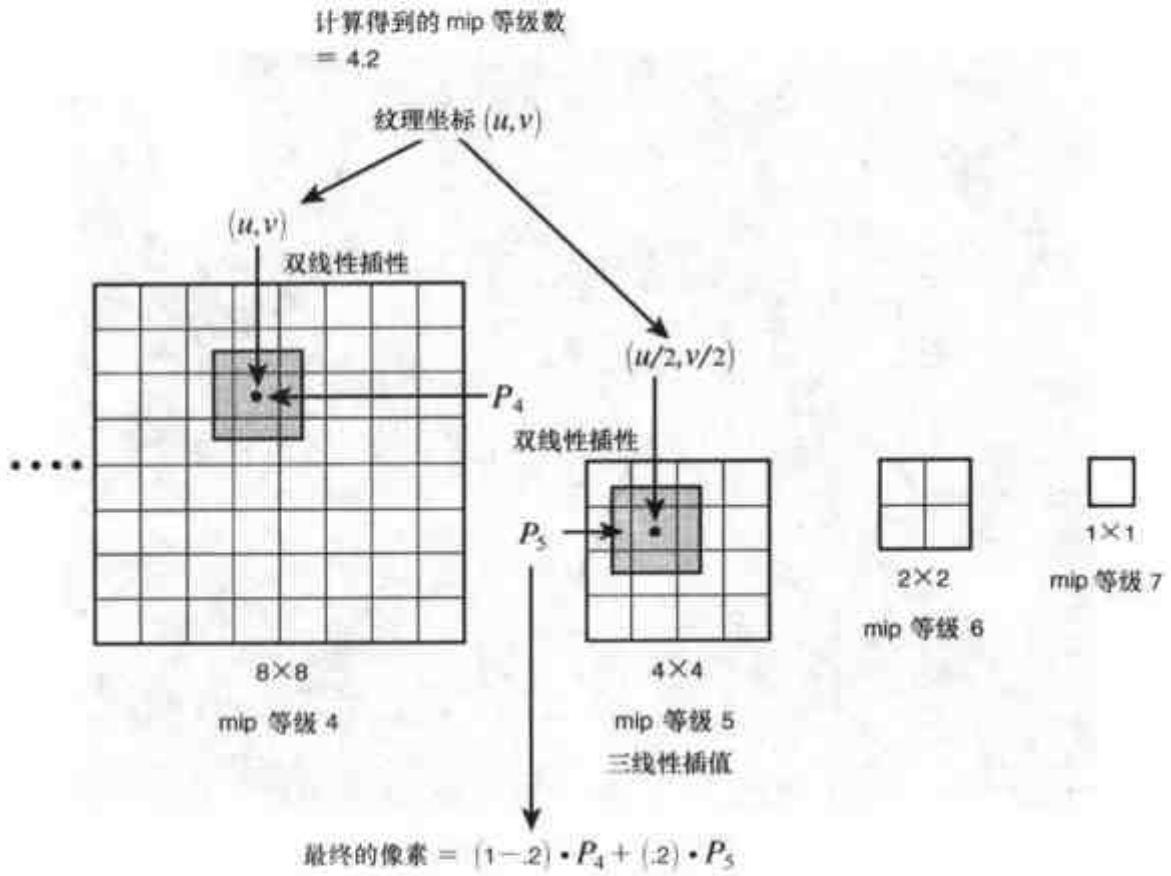


图 12.49　三线性滤波

12.8　多次渲染和纹理映射

令人难以置信的是，使用软件光栅化函数进行多次（multiple pass）渲染是可能的。可以这样考虑这一点：根据设置，当前能够在帧频为 15～30 帧/秒的情况下，每帧渲染大约 1000～3000 个多边形；进行两次渲染时只需将多边形数目减半即可。使用这种技术可以实现很多特殊效果，例如，可以这样来实现水中倒影效果：首先渲染正常的场景，然后渲染透明地倒转的场景，并将其与地平面（假定其表面是光洁的）进行 alpha 混合。

通过多次渲染，还可以实现阴影和其他特殊效果。然而，我们只允许最多执行两次渲染，即使场景非常简单，因为这样做将降低帧频。作为一个多次渲染范例，作者创建了一个场景，其中包含一些简单的几何形状，它们放置在一个平面上。为实现倒影效果，首先在不使用 z 缓存的情况下渲染地平面物体；然后将观察矩阵反转，并使用 alpha 混合和 z 缓存渲染倒影；最后使用正常的观察矩阵和 z 缓存对场景进行渲染，并将其视为完全不透明。最终的结果是，通过光洁表面得到了物体的倒影，如图 12.50 所示。

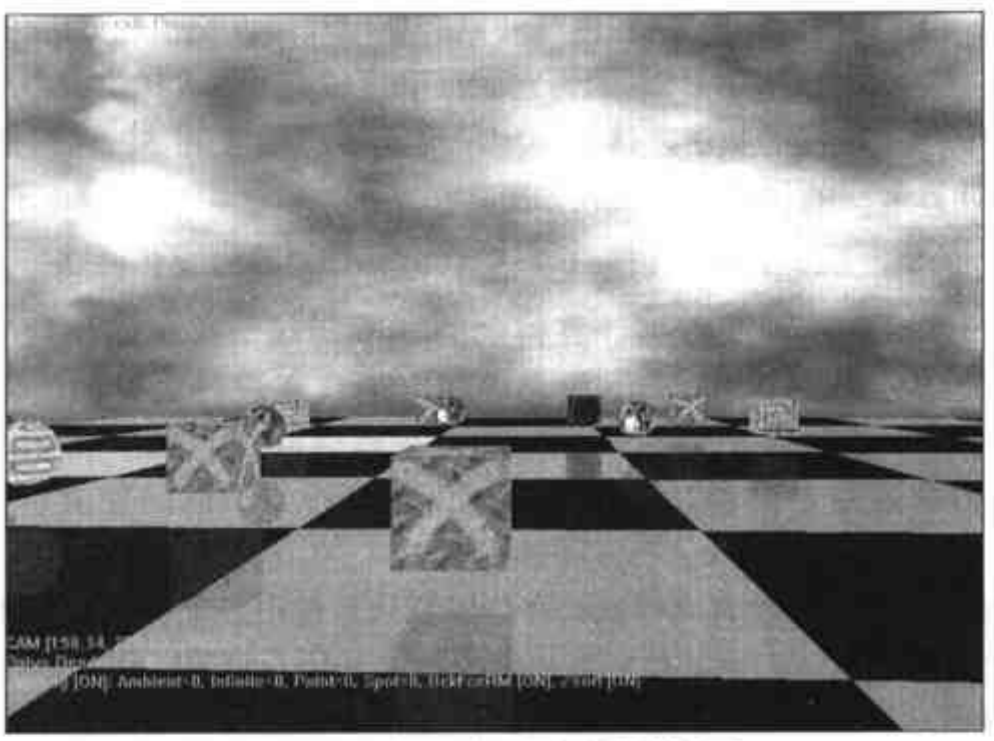


图 12.50　倒影演示程序的屏幕截图

该演示程序名为 DEMOII12_12.CPP|EXE。按箭头键可移动环境。要编译该程序，需要 DEMOII12_12.CPP|H、T3DLIB1-10.CPP|H 和 DirectX .LIB 文件。

12.9　使用单个函数来完成渲染工作

开始编写本章时，作者没想到会这么长，虽然作者已决定将有关光照和阴影的内容单列一章。本章的内容比较棘手，这进一步表明，即使是一个非常基本的软件 3D 引擎，也是一个复杂的项目；然而，现在引擎的功能已达到令人惊讶的程度。

现在，代码量已达到难以管理的程度，需要在渲染函数上增加一层，用于绘制渲染列表。当前，有支持排序、z 缓存、$1/z$ 缓存、mipmapping、alpha 混合、透视修正等的光栅函数，数量相当多；归根结底，这是完全不能接受的。图 12.51 说明了这一点。当然，如果您知道需要进行什么样的光栅化，则没有必要增加这一层；然而，现在创建演示程序时，手工调用这些光栅化函数变得过于复杂。

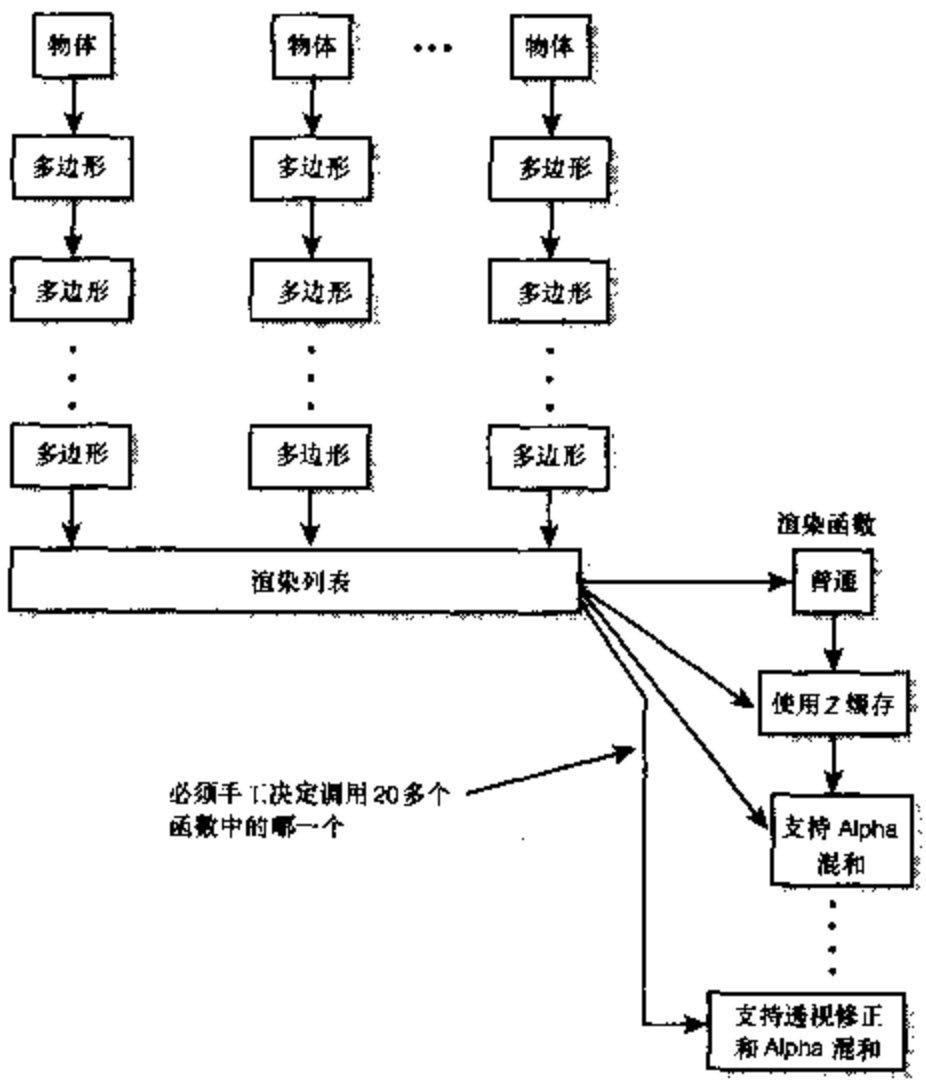


图 12.51　没有统一渲染入口的系统

12.9.1　新的渲染场境

为解决函数调用混乱的问题，作者创建了一个名为渲染场境的数据结构，可以像设置 DirectX 结构那样设置它。我们可以使用诸如视频缓存、z 缓存、渲染标记等信息来填充该数据结构，然后调用一个高级函数，该函数对渲染场境进行分析，并调用合适的光栅化函数来渲染三角形。图 12.52 说明了这一点。

这个数据结构的定义如下（它位于 T3DLIB10.H 中）：

```
typedef struct RENDERCONTEXTV1_TYP
{
int    attr;            // 渲染属性
RENDERLIST4DV2_PTR rend_list; // 指向要渲染的渲染列表的指针
UCHAR *video_buffer;     // 指向视频缓存的指针
```

```
int   lpitch;          // 视频缓存的跨距，单位为字节

UCHAR *zbuffer;        // 指向 z 缓存或 1/z 缓存的指针
int   zpitch;          // z 缓存或 1/z 缓存的跨距
int   alpha_override;  // 用于重新设置所有多边形的 alpha 值

int   mip_dist;            // 最大 mip 距离
int   texture_dist,        // 使用透视/仿射混合模式时，启用仿射纹理映射的距离

// 供以后扩展时使用
int   ival1, ivalu2;
float  fval1, fval2;
void  *vptr;

} RENDERCONTEXTV1, *RENDERCONTEXTV1_PTR;
```

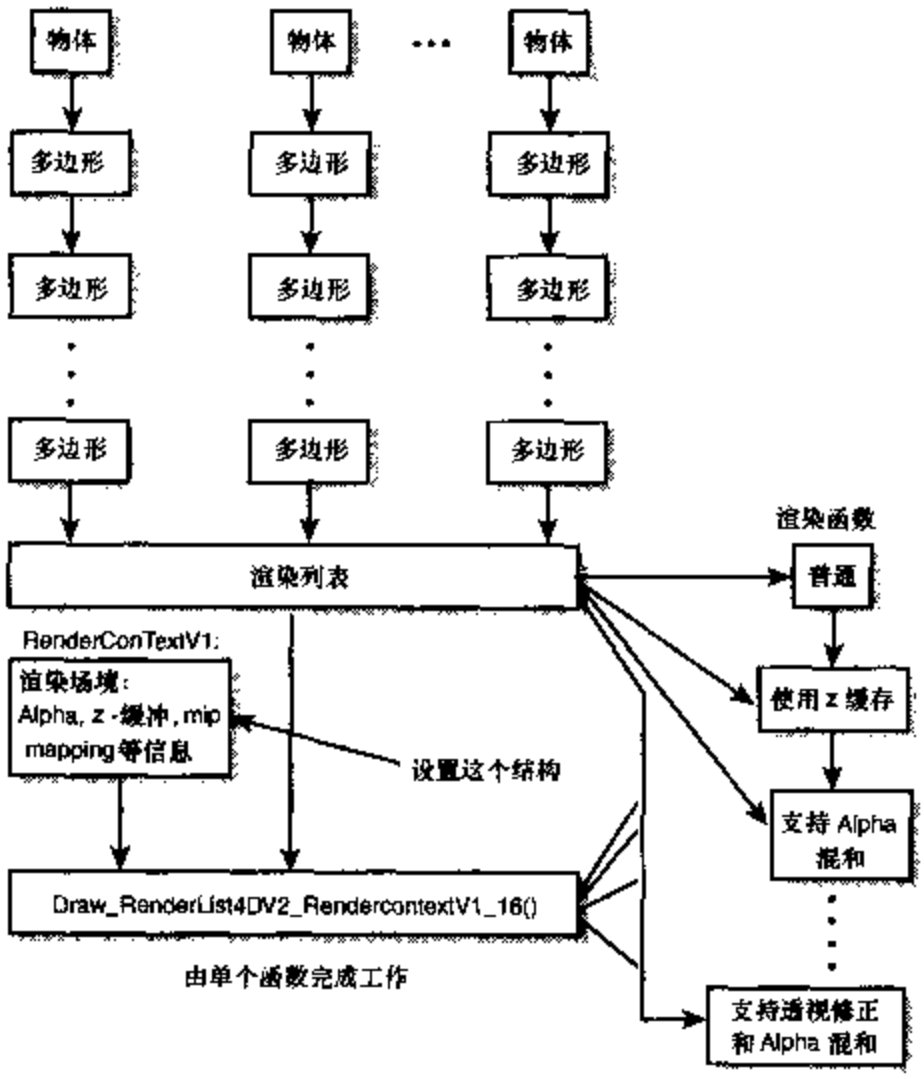


图 12.52　支持渲染场境的新系统

从本质上说，它只是将所有渲染函数的参数列表合并起来了，通过它可以设置渲染帧时需要的所有信息。虽然其中所有字段的含义和用途都是不言自明的，但这里还是要简单地介绍它们，以确保读者对其有正确的认识：

- attr——指定如何进行渲染，即使用哪些选项：*z* 缓存、不使用 *z* 缓存、alpha 混合、双线性插值、mipmapping 等。调用渲染函数之前，需要用逻辑运算符 OR 将所需的渲染选项组合起来，并使用它们来设置 attr。稍后将更详细地介绍。
- rend_list——一个指针，指向要渲染的渲染列表。
- video_buffer——指向视频缓存的指针，场景将被渲染到该视频缓存中。
- lpitch——video_buffer 的跨距，单位为字节。
- zbuffer——指向 *z* 缓存存储区域的指针。您可能使用 *z* 缓存或 1/*z* 缓存，但存储空间是相同的。
- zpitch——*z* 缓存的跨距，单位为字节。
- alpha_override——如果在渲染场境中启用了 alpha 混合，可以将这个变量设置为 0 到 NUM_ALPHA_LEVELS - 1，以重新设置整个场景的 alpha 值；如果不想这样做，则将其设置为-1。

- mip_mist——指定在什么样的距离范围内使用 mipmapping。在距离为 0 到 mip_dist 的范围内，将根据距离以线性方式选择 mip 等级。
- texture_dist 和 texture_dist2——在使用混合纹理映射方法时，它们用于选择纹理映射的控制范围。根据混合模式使用的是完美透视修正、线性分段和仿射纹理映射或者只是使用线性分段和仿射纹理映射，需要设置其中的一个或两个。稍后将做更详细的介绍。
- ival1 和 ival2——两个可随意使用的整数变量。
- fval1 和 fval2——两个可随意使用的浮点数变量。
- vptr——一个可随意使用的 void 指针。

12.9.2　设置渲染场境

接下来讨论如何设置渲染场境。首先，必须创建渲染场景（这通常是单个全局变量，但可能有多个渲染场境）：

```
RENDERCONTEXTV1 rc;
```

对于大型数据结构，使用类似于下面的代码清除其数据是个不错的主意：

```
memset(&rc,0,sizeof(rc));
```

渲染场境的 attr 是关键——只需使用所需的渲染选项设置它即可。渲染选项标记如下所示，它们是在 T3DLIB10.H 中定义的：

```
// 用于控制渲染函数状态的属性
// 每类控制标记都存储在一个 4 位字节中，这有助于以后扩展

// 不使用 z 缓存，按多边形在渲染列表中的顺序渲染它们
#define RENDER_ATTR_NOBUFFER                    0x00000001

// 使用 z 缓存
#define RENDER_ATTR_ZBUFFER                     0x00000002

// 使用 1/z 缓存
#define RENDER_ATTR_INVZBUFFER                  0x00000004

// 使用 mipmapping
#define RENDER_ATTR_MIPMAP                      0x00000010

// 启用 alpha 混合
#define RENDER_ATTR_ALPHA                       0x00000020

// 启用双线性滤波，但只适用于固定着色/仿射纹理映射函数
#define RENDER_ATTR_BILERP                      0x00000040

// 使用仿射纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE  0x00000100

// 使用完美透视修正纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT 0x00000200

// 使用线性分段透视纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR  0x00000400

// 根据距离混合使用仿射纹理映射和线性分段纹理映射
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1 0x00000800

// 还未实现
#define RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID2 0x00001000
```

接下来看一些设置渲染场境的例子。假设要将渲染场境设置为不使用深度缓存（预先对多边形进行 z 排序）。

1. 渲染场境设置范例 1

不使用 alpha 混合和 mipmapping，对于所有多边形都使用仿射纹理映射，设置如下：

```
// 设置渲染场境
rc.attr     = (RENDER_ATTR_NOBUFFER |
          RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE);

rc.video_buffer = back_buffer; // 设置视频缓存
rc.lpitch    = back_lpitch;
rc.mip_dist   = 0;
rc.zbuffer   = NULL;
rc.zpitch    = 0;
rc.rend_list  = &rend_list; // 渲染列表
rc.texture_dist = 0;
rc.alpha_override = -1;
```

2. 渲染场境设置范例 2

使用 z 缓存、alpha 混合和线性分段透视纹理映射：

```
// 设置渲染场境
rc.attr     = (RENDER_ATTR_ZBUFFER |
          RENDER_ATTR_ALPHA |
          RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR);

rc.video_buffer = back_buffer; // 设置视频缓存
rc.lpitch    = back_lpitch;
rc.mip_dist   = 0;
rc.zbuffer   = (UCHAR *)zbuffer.zbuffer; // 现在需要 z 缓存
rc.zpitch    = WINDOW_WIDTH*4;
rc.rend_list  = &rend_list;
rc.texture_dist = 0;
rc.alpha_override = -1;
```

3. 渲染场境设置范例 3

接下来设置一个这样的渲染场境：使用 mipmapping，最大 mip 距离为 2000。另外使用完美透视修正纹理映射和 $1/z$ 缓存。

```
// 设置渲染场境
rc.attr     = (RENDER_ATTR_INVZBUFFER |
          RENDER_ATTR_ALPHA |
          RENDER_ATTR_MIPMAP |
          RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT);

rc.video_buffer = back_buffer; // 设置视频缓存
rc.lpitch    = back_lpitch;
rc.mip_dist   = 2000; // 最大 mip 距离
rc.zbuffer   = (UCHAR *)zbuffer.zbuffer; // 1/z 缓存
rc.zpitch    = WINDOW_WIDTH*4;
rc.rend_list  = &rend_list;
rc.texture_dist = 0;
rc.alpha_override = -1;
```

注意：mip_dist 用于设置最大 mip 距离。多边形离视平面的距离小于 mip_dist 时，将根据距离选择 mip 等级。

4. 渲染场境设置范例 4

来看一个使用混合纹理映射方法的例子。这里使用混合模式 1：当多边形离视平面的距离小于 texture_dist 时，使用线性分段纹理映射；否则使用仿射纹理映射。我们将 texture_dist 设置为 500，这样，对于距离为 0～500 的多边形，将调用线性分段纹理映射函数；对于距离超过 500 的多边形，将使用仿射纹理映射。其他设置与前一个例子中相同。

```
// 设置渲染场境
rc.attr     = (RENDER_ATTR_INVZBUFFER |
          RENDER_ATTR_ALPHA |
          RENDER_ATTR_MIPMAP |
```

```
        RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1);

    rc.video_buffer = back_buffer; // 设置视频缓存
    rc.lpitch       = back_lpitch;
    rc.mip_dist     = 2000; // 最大 mip 距离
    rc.zbuffer      = (UCHAR *)zbuffer.zbuffer; // 1/z 缓存
    rc.zpitch       = WINDOW_WIDTH*4;
    rc.rend_list    = &rend_list;
    rc.texture_dist = 500;
    rc.alpha_override = -1;
```

警告：使用任何形式的透视修正纹理映射时，都必须使用 1/z 缓存，因为普通或 z 缓存光栅化函数不支持这种功能。

12.9.3　调用对渲染场境进行渲染的函数

设置好渲染场境后，需要对其进行渲染。我们需要一个这样的函数：根据渲染场境中的选项透明地完成渲染工作——决定对于每个多边形调用哪个光栅化函数。这个函数包含大量的条件逻辑，为让读者理解它，这里将列出其部分代码。

请注意，有更好的方法来完成这些工作。例如，在内循环中，对于每个多边形都执行一些有条件的运算。这些可以预先计算好，这样便可以通过函数指针来调用低级函数，尽可能地优化该函数。然而，如果这样做，读者将不知道工作是如何完成的，因此现在暂时没有这样进行优化。以后将加入函数指针支持，得到一个更为优化、速度更快的版本。

这个函数的代码太长（15~30 页），这里不能全部列出，而只列出其中的一部分。这个函数通过首先判断调用者请求不使用深度缓存、使用 z 缓存还是 1/z 缓存，对条件逻辑进行了一些优化。下面是处理不使用深度缓存情形的代码，处理其他情形的代码与此类似，为节省篇幅，这里没有列出它们。

```
void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16(RENDERCONTEXTV1_PTR rc)
{
// 该函数根据渲染场境对渲染列表进行渲染
// 渲染场境中包含渲染时需要的所有信息：
// 是否使用 z 缓存、1/z 缓存、alpha 混合、mipmapping、透视修正、双线性插值等
// 通过该函数能够调用所有的光栅化函数
// 但比分别调用这些函数更好

POLYF4DV2 face; // 用于存储要渲染的多边形
int alpha;   // alpha 值

// 需要首先判断各种条件，以最大限度地减少遍历渲染列表时的计算量
// 首先判断渲染方式：不使用深度缓存、使用 z 缓存还是使用 1/z 缓存

if (rc->attr & RENDER_ATTR_NOBUFFER) ////////////////////////
{
// 根本不使用深度缓存

for (int poly=0; poly < rc->rend_list->num_polys; poly++)
  {
  // 仅当多边形没有被剔除和裁剪掉、处于活动状态且可见时才对其进行渲染
  if (!(rc->rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_ACTIVE) ||
     (rc->rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_CLIPPED ) ||
     (rc->rend_list->poly_ptrs[poly]->state & POLY4DV2_STATE_BACKFACE) )
    continue; // 进入下一个多边形

  // 检测是否在渲染场境中设置了 alpha 值
  if (rc->alpha_override>= 0)
    {
    // 将 alpha 设置为渲染场境中的 alpha 值
    alpha = rc->alpha_override;
    } // end if
  else
    {
    // 提取 alpha 值
```

```
    alpha = ((rc->rend_list->poly_ptrs[poly]->color & 0xff000000) >> 24);
    } // end else


  // 首先检测多边形是否带纹理
  // 因为带纹理的多边形可能使用固定或恒定着色，需要据此调用不同的光栅化函数
  if (rc->rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
    {
    // 设置顶点信息
    face.tvlist[0].x = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].z;
    face.tvlist[0].u0 = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].u0;
    face.tvlist[0].v0 = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].v0;

    face.tvlist[1].x = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].z;
    face.tvlist[1].u0 = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].u0;
    face.tvlist[1].v0 = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].v0;

    face.tvlist[2].x =  (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y =  (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z =  (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].z;
    face.tvlist[2].u0 =  (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].u0;
    face.tvlist[2].v0 = (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].v0;

    // 检查多边形的 mipmap 属性
    if (rc->rend_list->poly_ptrs[poly]->attr & POLY4DV2_ATTR_MIPMAP)
     {
     // 检查渲染场境的 mipmap 属性
     if (rc->attr & RENDER_ATTR_MIPMAP)
       {
       // 确定使用哪个 mip 纹理

       // 首先计算该多边形的 mip 纹理链的 mip 等级数
       int tmiplevels = logbase2ofx(((BITMAP_IMAGE_PTR *)
        (rc->rend_list->poly_ptrs[poly]->texture))[0]->width];

       // 根据最大 mip 距离和多边形的 z 坐标确定使用哪个 mip 等级
       // 这里使用多边形一个顶点的 z 坐标，读者可能想使用平均 z 值
       int miplevel = (tmiplevels *
         rc->rend_list->poly_ptrs[poly]->tvlist[0].z / rc->mip_dist);

       // 截取计算得到的 mip 等级
       if (miplevel > tmiplevels) miplevel = tmiplevels;

       // 根据 mip 等级选择相应的纹理
       face.texture = ((BITMAP_IMAGE_PTR *)
         (rc->rend_list->poly_ptrs[poly]->texture))[miplevel];

       // 根据 mip 等级，执行相应次数的"纹理坐标除以 2"运算
       for (int ts = 0; ts < miplevel; ts++)
         {
         face.tvlist[0].u0*=.5;
         face.tvlist[0].v0*=.5;

         face.tvlist[1].u0*=.5;
         face.tvlist[1].v0*=.5;

         face.tvlist[2].u0*=.5;
         face.tvlist[2].v0*=.5;
        } // end for

     } // end if mipmmaping enabled globally
   else // 渲染场境的 mipmap 属性没有被设置
     {
```

```
    //用户没有请求使用 mipmaping，因此使用 0 级别纹理
    face.texture = ((BITMAP_IMAGE_PTR *)
           (rc->rend_list->poly_ptrs[poly]->texture))[0];

    } // end else

  } // end if
 else
  {
  face.texture = rc->rend_list->poly_ptrs[poly]->texture;
  } // end if

 // 使用固定着色?
 if (rc->rend_list->poly_ptrs[poly]->attr &
   POLY4DV2_ATTR_SHADE_MODE_CONSTANT)
  {
  // 使用固定着色绘制带纹理的三角形

  if ((rc->attr & RENDER_ATTR_ALPHA) &&
    ((rc->rend_list->poly_ptrs[poly]->attr &
     POLY4DV2_ATTR_TRANSPARENT) || rc->alpha_override>=0) )
    {
    // 使用 alpha 混合

    // 决定调用哪个纹理映射函数
    if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE)
     {
     Draw_Textured_Triangle_Alpha16(&face, rc->video_buffer,
                    rc->lpitch, alpha);
     } // end if
    else
    if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT)
     {
     // 还不支持
     Draw_Textured_Triangle_Alpha16(&face, rc->video_buffer,
                    rc->lpitch, alpha);
     } // end if
    else
    if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR)
     {
     // 还不支持
     Draw_Textured_Triangle_Alpha16(&face, rc->video_buffer,
                    rc->lpitch, alpha);
     } // end if
    else
    if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1)
     {
     // 将 z 值与透视修正纹理映射过渡点进行比较
     if (rc->rend_list->poly_ptrs[poly]->tvlist[0].z >
       rc-> texture_dist)
       {
       // 使用仿射纹理映射
       Draw_Textured_Triangle_Alpha16(&face, rc->video_buffer,
                      rc->lpitch, alpha);
       } // end if
     else
       {
       // 使用线性分段纹理映射
       // 还不支持
       Draw_Textured_Triangle_Alpha16(&face, rc->video_buffer,
                      rc->lpitch, alpha);
       } // end if

    } // end if

  } // end if
else
```

```
{
// 不使用 alpha 混合
// 使用哪个纹理映射函数?
if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE)
 {
 Draw_Textured_Triangle2_16(&face, rc->video_buffer,
                rc->lpitch);
 } // end if
else
if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT)
 {
 // 还不支持
 Draw_Textured_Triangle2_16(&face, rc->video_buffer,
                rc->lpitch);
 } // end if
else
if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR)
 {
 // 还不支持
 Draw_Textured_Triangle2_16(&face, rc->video_buffer,
                rc->lpitch);
 } // end if
else
if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1)
 {
 // 将 z 值同透视纹理映射过渡点进行比较
 if (rc->rend_list->poly_ptrs[poly]->tvlist[0].z >
   rc-> texture_dist)
   {
   // 使用仿射纹理映射
   Draw_Textured_Triangle2_16(&face, rc->video_buffer,
                    rc->lpitch);
   } // end if
 else
   {
   // 使用线性分段纹理映射
   // 还不支持
   Draw_Textured_Triangle2_16(&face, rc->video_buffer,
                    rc->lpitch);
   } // end if

  } // end if

 } // end if
else
if (rc->rend_list->poly_ptrs[poly]->attr &
  POLY4DV2_ATTR_SHADE_MODE_FLAT)
 {
 // 使用恒定着色
 face.lit_color[0] = rc->rend_list->poly_ptrs[poly]->lit_color[0];

 if ((rc->attr & RENDER_ATTR_ALPHA) &&
   ((rc->rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_TRANSPARENT) ||
    rc->alpha_override>=0) )
   {
   // alpha 版本

   // 使用哪个纹理映射函数?
   if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE)
    {
    Draw_Textured_TriangleFS_Alpha16(&face, rc->video_buffer,
                     rc->lpitch, alpha);
    } // end if
   else
```

```
if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT)
 {
 // 还不支持
 Draw_Textured_TriangleFS_Alpha16(&face, rc->video_buffer,
                  rc->lpitch, alpha);
 } // end if
else
if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR)
 {
 // 还不支持
 Draw_Textured_TriangleFS_Alpha16(&face, rc->video_buffer,
                  rc->lpitch, alpha);
 } // end if
else
if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1)
 {
 // 将 z 值同透视纹理映射过渡点进行比较
 if (rc->rend_list->poly_ptrs[poly]->tvlist[0].z >
    rc-> texture_dist)
    {
    // 使用仿射纹理映射
    Draw_Textured_TriangleFS_Alpha16(&face, rc->video_buffer,
                     rc->lpitch, alpha);
    } // end if
 else
    {
    // 使用线性分段纹理映射
    // 还不支持
    Draw_Textured_TriangleFS_Alpha16(&face,
                     rc->video_buffer, rc->lpitch, alpha);
    } // end if

  } // end if
  } // end if
else
  {
  // 不使用 alpha 混合
  // 调用哪个纹理映射函数?
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE)
   {
   Draw_Textured_TriangleFS2_16(&face, rc->video_buffer,
                    rc->lpitch);
   } // end if
  else
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT)
   {
   // 还不支持
   Draw_Textured_TriangleFS2_16(&face, rc->video_buffer,
                    rc->lpitch);
   } // end if
  else
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR)
   {
   // 还不支持
   Draw_Textured_TriangleFS2_16(&face, rc->video_buffer,
                    rc->lpitch);
   } // end if
  else
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1)
   {
   // 将 z 值同透视纹理映射过渡点进行比较
   if (rc->rend_list->poly_ptrs[poly]->tvlist[0].z >
      rc-> texture_dist)
      {
      // 使用仿射纹理映射
      Draw_Textured_TriangleFS2_16(&face, rc->video_buffer,
```

```
                        rc->lpitch);
          } // end if
        else
          {
          // 使用线性分段纹理映射
          // 还不支持
          Draw_Textured_TriangleFS2_16(&face, rc->video_buffer,
                        rc->lpitch);
          } // end if

        } // end if

      } // end if

    } // end else
  else
    {
    // 必然使用 Gouraud 纹理映射
    face.lit_color[0] = rc->rend_list->poly_ptrs[poly]->lit_color[0];
    face.lit_color[1] = rc->rend_list->poly_ptrs[poly]->lit_color[1];
    face.lit_color[2] = rc->rend_list->poly_ptrs[poly]->lit_color[2];

    if ((rc->attr & RENDER_ATTR_ALPHA) &&
      ((rc->rend_list->poly_ptrs[poly]->attr &
       POLY4DV2_ATTR_TRANSPARENT) ||
       rc->alpha_override>=0) )
      {
      // 使用 alpha 混合

      // 应调用哪个纹理映射函数?
      if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE)
       {
       Draw_Textured_TriangleGS_Alpha16(&face, rc->video_buffer,
                        rc->lpitch, alpha);
    } // end if
    else
    if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT)
    {
    // 还不支持
    Draw_Textured_TriangleGS_Alpha16(&face, rc->video_buffer,
                      rc->lpitch, alpha);
    } // end if
    else
    if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR)
    {
    // 还不支持
    Draw_Textured_TriangleGS_Alpha16(&face, rc->video_buffer,
                      rc->lpitch, alpha);
    } // end if
    else
    if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1)
     {
     // 将 z 值同透视纹理映射过渡点进行比较
     if (rc->rend_list->poly_ptrs[poly]->tvlist[0].z >
       rc-> texture_dist)
       {
       // 使用仿射纹理映射
       Draw_Textured_TriangleGS_Alpha16(&face,
                        rc->video_buffer, rc->lpitch, alpha);
       } // end if
     else
       {
       // 使用线性分段纹理映射
       // 还不支持
       Draw_Textured_TriangleGS_Alpha16(&face,
                        rc->video_buffer, rc->lpitch, alpha);
       } // end if
```

```
    } // end if

  } // end if
else
  {
  // 不使用 alpha 混合
  // 应调用哪个纹理映射函数?
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE)
   {
   Draw_Textured_TriangleGS_16(&face, rc->video_buffer,
                rc->lpitch);
   } // end if
  else
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_CORRECT)
   {
   // 还不支持
   Draw_Textured_TriangleGS_16(&face, rc->video_buffer,
                rc->lpitch);
   } // end if
  else
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_LINEAR)
   {
   // 还不支持
   Draw_Textured_TriangleGS_16(&face, rc->video_buffer,
                rc->lpitch);
   } // end if
  else
  if (rc->attr & RENDER_ATTR_TEXTURE_PERSPECTIVE_HYBRID1)
   {
   // 将 z 值同透视纹理映射过渡点进行比较
   if (rc->rend_list->poly_ptrs[poly]->tvlist[0].z >
     rc-> texture_dist)
     {
     // 使用仿射纹理映射
     Draw_Textured_TriangleGS_16(&face, rc->video_buffer,
                  rc->lpitch);
     } // end if
    else
     {
     // 使用线性分段纹理映射
     // 还不支持
     Draw_Textured_TriangleGS_16(&face, rc->video_buffer,
                  rc->lpitch);
     } // end if

    } // end if

   } // end if

  } // end else

  } // end if
else
if ((rc->rend_list->poly_ptrs[poly]->attr &
  POLY4DV2_ATTR_SHADE_MODE_FLAT) ||
  (rc->rend_list->poly_ptrs[poly]->attr &
   POLY4DV2_ATTR_SHADE_MODE_CONSTANT) )
  {
  // 使用固定着色
  face.lit_color[0] = rc->rend_list->poly_ptrs[poly]->lit_color[0];

  // 设置顶点信息
  face.tvlist[0].x = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].x;
  face.tvlist[0].y = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].y;
  face.tvlist[0].z = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].z;
```

```
    face.tvlist[1].x =  (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y =  (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z =  (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].z;

    face.tvlist[2].x = (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y = (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z = (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].z;

    // 使用基本恒定着色光栅化函数绘制三角形

    // 检测透明性
    if ((rc->attr & RENDER_ATTR_ALPHA) &&
       ((rc->rend_list->poly_ptrs[poly]->attr &
        POLY4DV2_ATTR_TRANSPARENT) || rc->alpha_override>=0) )
     {
     Draw_Triangle_2D_Alpha16(&face, rc->video_buffer, rc->lpitch,alpha);
     } // end if
    else
     {
     Draw_Triangle_2D3_16(&face, rc->video_buffer, rc->lpitch);
     } // end if

    } // end if
  else
  if (rc->rend_list->poly_ptrs[poly]->attr &
    POLY4DV2_ATTR_SHADE_MODE_GOURAUD)
    {
    // 设置顶点信息
    face.tvlist[0].x = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].x;
    face.tvlist[0].y = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].y;
    face.tvlist[0].z = (float)rc->rend_list->poly_ptrs[poly]->tvlist[0].z;
    face.lit_color[0] = rc->rend_list->poly_ptrs[poly]->lit_color[0];

    face.tvlist[1].x = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].x;
    face.tvlist[1].y = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].y;
    face.tvlist[1].z = (float)rc->rend_list->poly_ptrs[poly]->tvlist[1].z;
    face.lit_color[1] = rc->rend_list->poly_ptrs[poly]->lit_color[1];

    face.tvlist[2].x = (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].x;
    face.tvlist[2].y = (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].y;
    face.tvlist[2].z = (float)rc->rend_list->poly_ptrs[poly]->tvlist[2].z;
    face.lit_color[2] = rc->rend_list->poly_ptrs[poly]->lit_color[2];

    // 使用 Gouraud 着色绘制三角形
    // 检测透明性
     if ((rc->attr & RENDER_ATTR_ALPHA) &&
       ((rc->rend_list->poly_ptrs[poly]->attr &
       POLY4DV2_ATTR_TRANSPARENT) || rc->alpha_override>=0) )
     {
     Draw_Gouraud_Triangle_Alpha16(&face, rc->video_buffer,
                    rc->lpitch,alpha);
     } // end if
    else
     {
     Draw_Gouraud_Triangle2_16(&face, rc->video_buffer, rc->lpitch);
     } // end if

    } // end if gouraud

  } // end for poly

} // end if RENDER_ATTR_NOBUFFER

else
if (rc->attr & RENDER_ATTR_ZBUFFER)//////////////////
{
// 使用 z 缓存时的代码块
```

```
} // end if RENDER_ATTR_ZBUFFER
else
if (rc->attr & RENDER_ATTR_INVZBUFFER)
{
// 使用 1/z 缓存时的代码块

} // end if RENDER_ATTR_INVZBUFFER

} // end Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16
```

上述代码检查各种渲染条件，并据此做相应的处理。请注意其中处理 alpha 混合、mipmapping 和混合纹理映射的代码。不可否认，与直接调用合适的光栅化函数相比，这样做的速度更慢，因为调用高级函数将使性能降低大约 1～3%，但以后我们将把这种损失夺回来，甚至再赚一些。与以手工方式从大量的光栅化函数中选择一个相比，这样做好得多。下面使用前面设置的渲染场境之一来调用该函数：

```
// 渲染场境
Draw_RENDERLISTV2_RENDERCONTEXTV1_16(&rc);
```

这样做是否值得呢？作者认为值得。没有专门演示这个函数的程序，因为可演示的内容太少；但如果读者查看之前的演示程序，将发现它们使用的就是这个函数。

12.10　总　　结

这又是篇幅很长的一章。它涵盖的内容看似不多，但真正去介绍时却急剧膨胀。另外，本章编写的代码比任何一章都多，但都是值得的。现在，引擎的功能非常丰富，可以使用它来编写游戏了。我们添加了透视修正纹理映射、alpha 混合、mipmapping、1/z 缓存等功能，进行了大量的优化，并重新编写了引擎的光栅化接口，以提供单个接口。另外，我们还给.COB 模型加载函数和地形生成函数新增了一些特性。最后，还提供了一些优秀的演示程序，为读者尝试使用这些功能提供了良好的起点。本书旨在让读者深入理解 3D 算法及其实时实现，而不是教授尖端技术。现在的引擎很不错——作者运行一个演示程序时，刚好有人进来，看到如此好的光照和纹理映射效果后，他还以为使用的是硬件引擎呢！事实上，如果本书就此结束，也足以让读者忙活一阵子。但既然已经进行到了这里，为何半途而废呢？

作者惟一感到遗憾的是，系统开始出现 bug 了。代码如此之多，作者几乎没有时间进行详细的测试。当然，本书的演示程序只用作教学辅助手段，读者可以编写自己的引擎，但是发现缺陷总是一件令人不快的事情。作者将尽可能排除这些 bug。当前，让作者烦恼的主要问题是，3D 裁剪函数好像不能妥善地处理大型多边形，不过没有什么东西是十全十美的。下一章将介绍空间划分技术，并开始处理世界几何体。

第 13 章　空间划分和可见性算法

本章介绍空间划分算法，如二元空间划分、包围体层次结构（bounding hierarchical volume）、八叉树、入口（portal）等。另外，还将讨论通用的可见性算法，如潜在可见集（potentially visible set）和遮掩剔除（occlusion culling）。本章内容的技术性非常强，同时篇幅有限，因此只实现其中的两种技术，其中包括 BSP 树。然而，这足以让读者忙活一阵子。下面是本章将介绍的内容：

- 新的游戏引擎库模块；
- 空间划分和可见面判定简介；
- 二元空间划分；
- 潜在可见集合；
- 入口；
- 包围体层次结构和八叉树；
- 遮掩剔除。

13.1　新的游戏引擎模块

本章需要的代码很多，可将其作为一个单独的库模块，它是 T3DLIB11。编译本章的程序时，需要程序的主.CPP 文件、DirectX .LIB 文件和新增的库模块：

- T3DLIB11.CPP——空间划分技术等的 C/C++源文件。
- T3DLIB11.H——头文件。

注意：还必须链接 T3DLIB1-10.CPP|H。

本应该列出源文件和头文件的代码，但篇幅有限，不能这样做。然而，后面将列出本章涉及的每个函数和数据结构——完整的源代码或函数原型。

13.2　空间划分和可见面判定简介

空间划分和可见面判定之间是紧密相联的。3D 世界通常充斥着物体、地形、室内环境（interior）和室外环境（exterior）。问题在于，这些模型网格代表的多边形即使没有数百万（甚至数亿）个，也有数千个。在此之前，我们竭尽全力地找出各种方法，将尽可能多的多边形从渲染流水线中删除。当前，我们的系统在很大程度上说是基于物体的，其中每个物体都是一组多边形。我们首先对这些物体进行变换，然后将其插入到全局性渲染列表中，进而执行其他的变换，如光照处理、裁剪、投影和光栅化等。然而，本书前面指出过，可以使用包围球将位于视景体外的物体剔除，并在将物体传递给渲染流水线之前根据视景体对其进行裁剪，如图 13.1 所示。

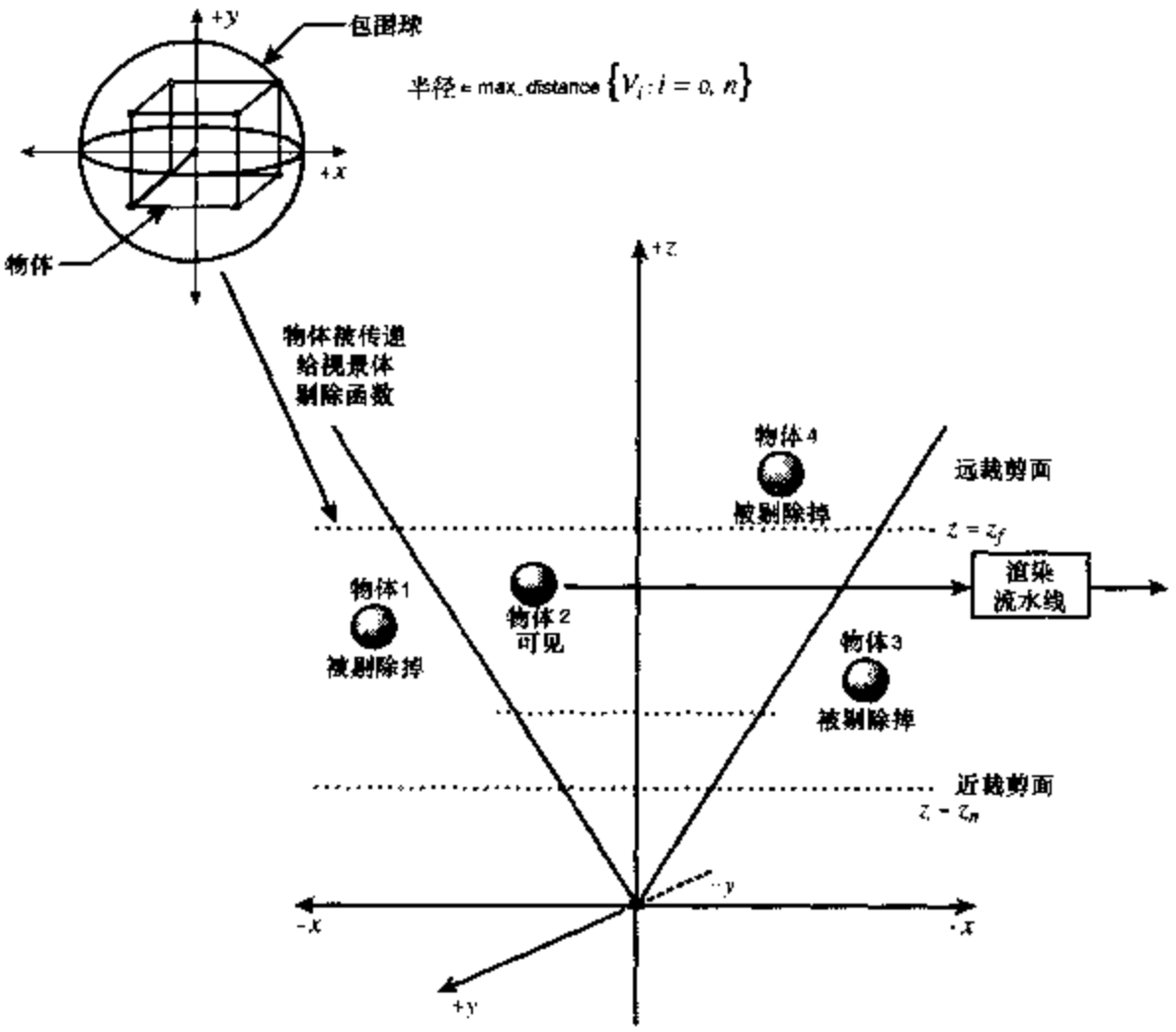


图 13.1　包围球剔除技术

然而，使用这种粗糙的技术时，很多物体并不能被剔除掉，进而得以传递到流水线的下游。例如，对于很长的物体、包含多个部分的物体等，包围球技术的效果并不好。但是，我们没有气馁，通过背面消除将没有背对观察者的多边形从流水线中删除，如图 13.2 所示。

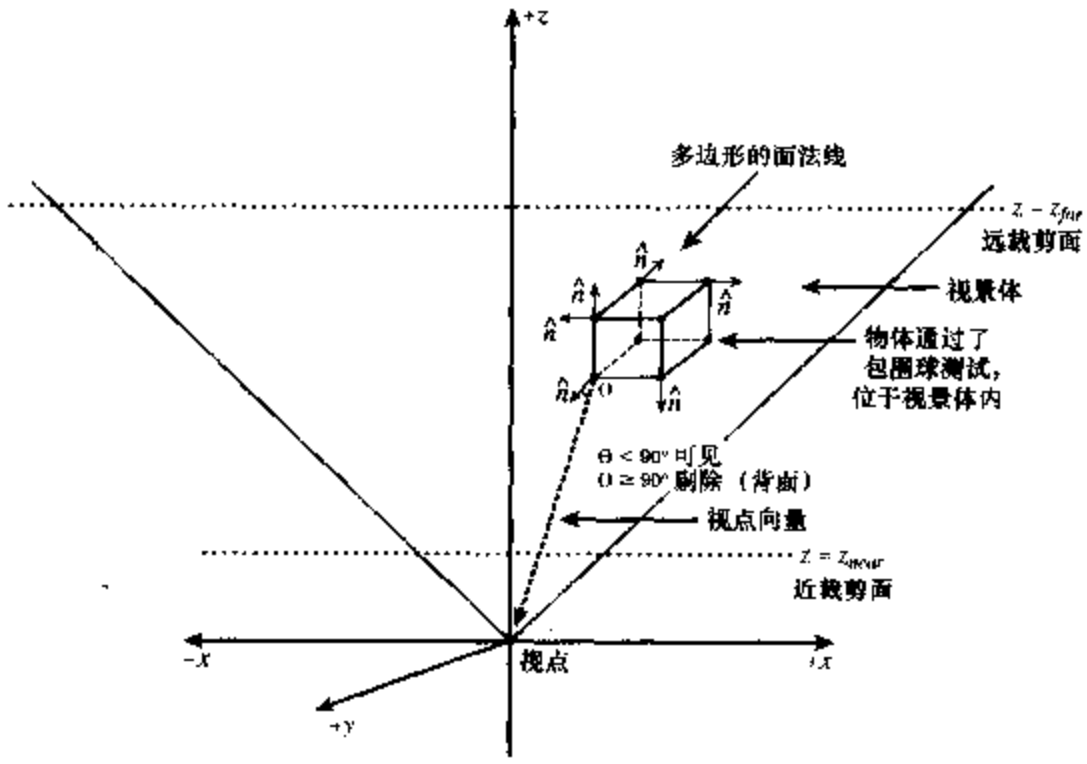


图 13.2　背面剔除

最后，将没有被剔除的多边形传递给 3D 裁剪函数，在物体空间中根据视景体对其进行裁剪，并在光栅化期间执行 2D 空间裁剪。裁剪处理是一种基于多边形的剔除，可以将整个多边形剔除掉，如图 13.3 所示。

在很大程度上说，对于基于物体的且物体不多的室外场景来说，这种系统的效果不错，甚至可用于创建粗糙的地形演示程序。关键之处在于，没有利用任何几何信息和常识，这正是本章的主题。我们感兴趣的是下列两个主要问题的解决方案：

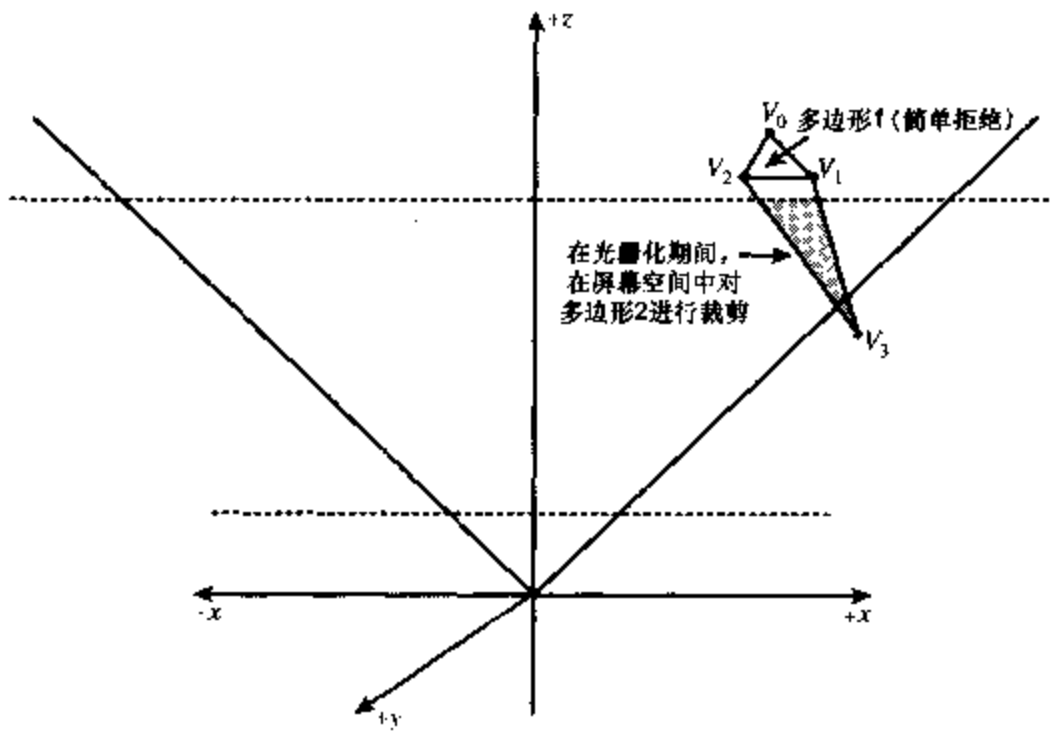


图 13.3　多边形裁剪

问题 1：如何对空间/物体进行划分（可能使用预处理或其他假设），以便能够快速地将大部分场景剔除掉，以帮助渲染和可见性判定？

问题 2：有办法在游戏中根据任何给定的视点确定哪些多边形可能是可见的，进而只对这些多边形进行处理吗？

这两个问题有些地方是重叠的，它们都属于可见性判定、遮掩和空间划分的范畴，这些正是本章要讨论的内容。

在计算机图形学中，最大的问题不是光栅化、光照处理或本书前面讨论过的众多其他主题。这些任务当然很重要——在很大程度上说，使用硬件能够解决这些问题。然而，硬件还不能解决本章将讨论的问题，这些问题也许永远得不到一劳永逸的解决。换句话说，在多边形数量不多的情况下，某种技术可能很管用；但当多边形数量增加 10 倍后，可能需要使用新的技术。直线绘制算法就是直线绘制算法——它不关心我们绘制的是房子还是宇宙飞船，但空间划分和可见性算法确实关心其处理的数据集的类型和大小。随着游戏世界的增大，数据集也将不断增大。因此可见性判定和剔除操作生死攸关，因为不渲染是最快的绘制方式。

例如，要创建一个 1024×1024 的地形，可以使用我们的地形生成函数，但这样将有大量的多边形被传递给剔除函数和裁剪函数，因为整个地形是一个物体。一种更好的技术是，创建一个系列地形，每个地形基于一个 32×32 的单元格，这样每个单元格独立的地形，它包含 32×32 个分片（32×32×2 个三角形，因为每个分片由两个三角形组成）。然后，将每个地形作为单个物体进行剔除，如图 13.4 所示。

是否可以做得更好呢？将地形分片按 8×8 或 16×16 的方式分组，然后在每组周围加上包围矩形如何？也就是说，可以创建层次型数据结构，用于存储世界数据，这样在剔除或可见性判定阶段，便可以立刻排除大量的数据。对于大型地形、室内环境和太空环境来说，这样做是完全必要的。可以建立包含 100 000 000 个星球系的银河系模型吗？这是可能的：可以存储每个星球系中每个星球的位置；存储一种包含一组参数的算法，它能够动态地生成任意详细程度的星球地形；可以创建每个星球上建筑物、植被等记录。然而，不能在每次迭代中遍历全部 100 000 000 个星球系，并且判断是否需要渲染它们。但是，使用空间划分系统后，只需要几次迭代便可完成这项工作——信不信由您！

本章将介绍多种空间划分技术，首先介绍二元空间划分，它是最常用、功能最强大的算法之一，本章将实现这种算法。为何需要使用这种技术呢？因为处理大型数据集时，不能再在流水线中排除多边形了。

另外，在第一人称射击游戏中，玩家在 99%的时间内都位于由数百个房间组成的游戏关卡中，但只能看到当前房间的几何体（当然，房间可能有窗户）。在很大程度上说，不应渲染其他的房间，因此必须找出处理这类问题的技术和算法。图 13.5 说明了这种可见性问题。

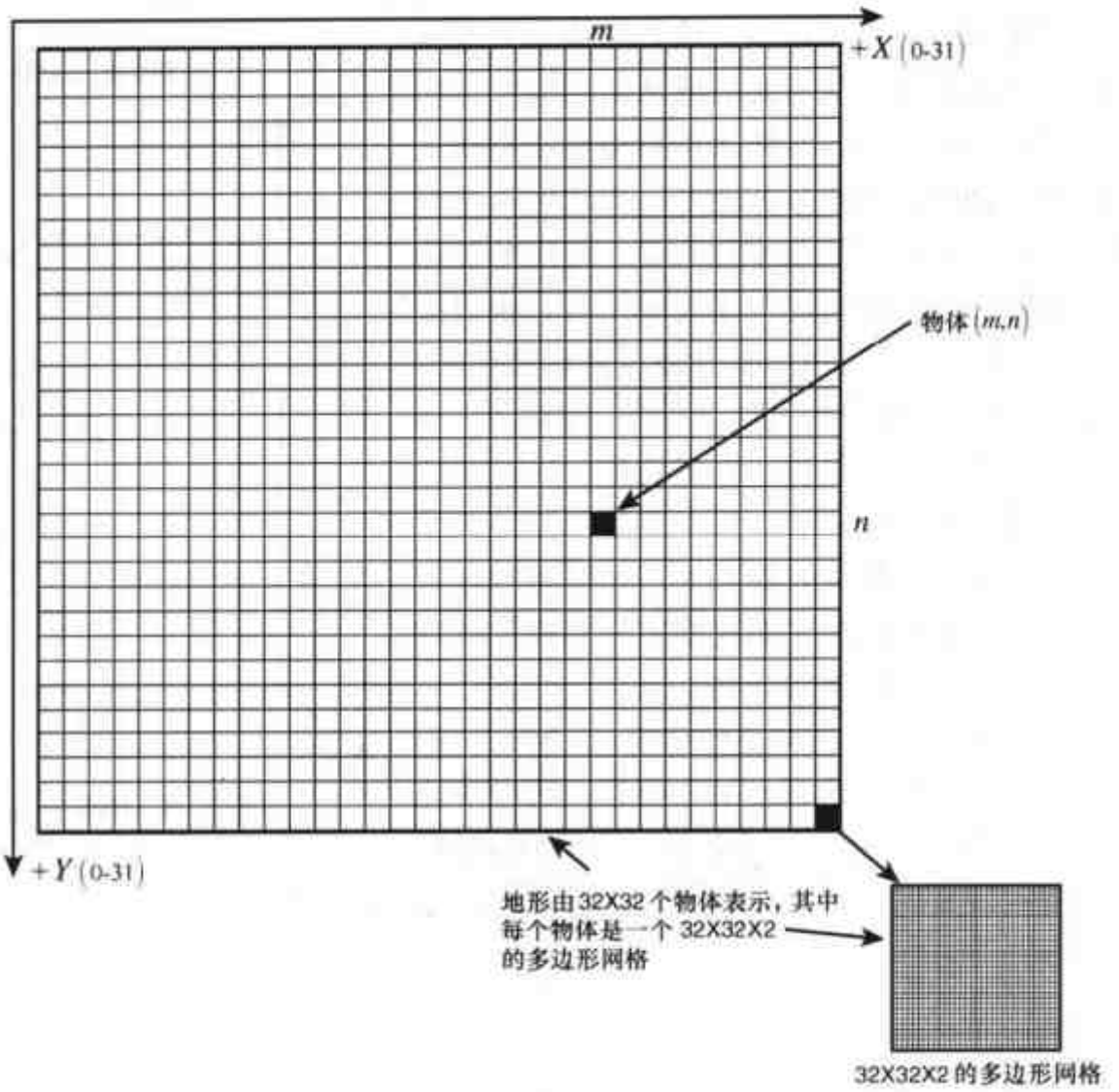


图 13.4　物体的层次型表示

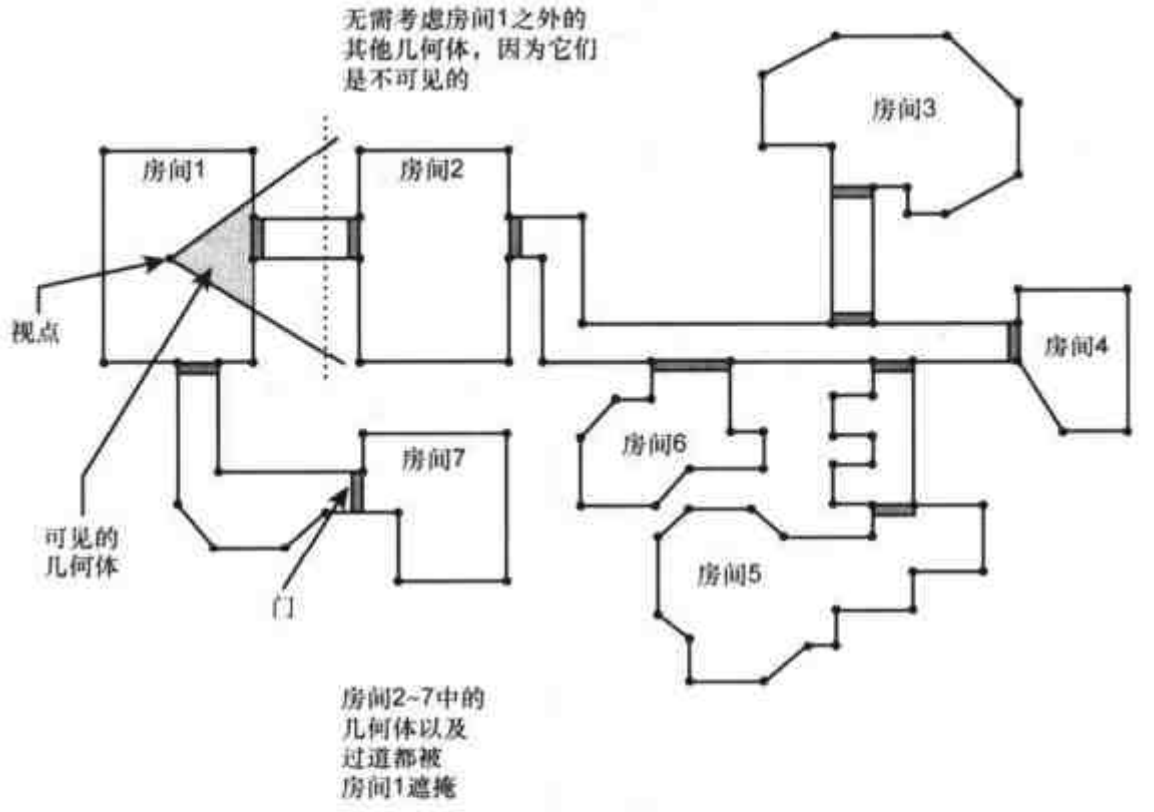


图 13.5　可见性问题

本章将讨论多种技术，有些只从理论上进行探讨，一些是从实际应用的角度探讨，这样读者将能够处理室内/室外渲染涉及的任何问题，至少知道需要做什么以及从哪里着手。

13.3　二元空间划分

二元空间划分（binary space partition，BSP）是一种 3D 空间划分算法，以初始空间（initial space）和离线计算时间（基于一些几何约束条件）换取更高的运行性能。从本质上说，BSP 将一组多边形作为输入，

通过一种递归算法，创建一个二叉树结构。这种结构具有很多有趣的特性，让我们能够按正确的从前到后或从后到前的顺序遍历多边形。我们还可以使用这种二叉树来进行碰撞检测或大规模的剔除操作。Doom、Quake 和众多其他第一人称射击游戏都是基于这种数据结构的。BSP 树的基本原理是，使用分隔面将空间划分成凸形子空间。分隔面可能与坐标轴平行，也可能与多边形共面，但关键之处在于，几何体本身被用来划分空间。

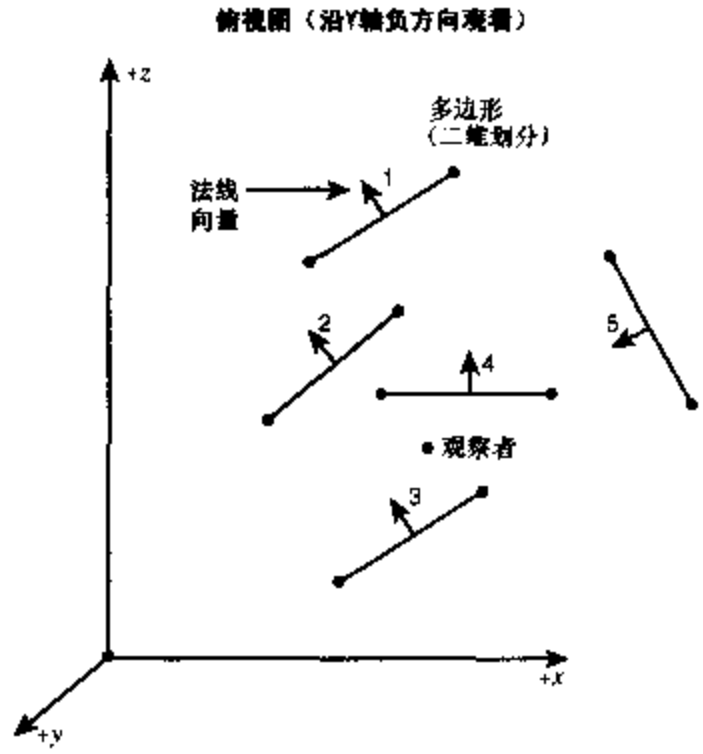


图 13.6　BSP 要解决的基本问题

BSP 好得令人难以置信，下面介绍其工作原理。请看图 13.6 所示的 x-z 平面上的俯视图，其中包含一组多边形，这些多边形可能构成了一个游戏关卡的几何形状。每个多边形都定义了一个平面，它位于该平面上。正如读者看到的，如果将观察者置于图中的任何位置，将不难根据给定的观察方向和视景体确定多边形的绘制顺序，进而正确地渲染场景。

例如，在图 13.7 中，从后到前的渲染顺序为 a、b、c、d、e 和 f。前面介绍的很多方法，如画家算法和 z 缓存，可以解决这种问题。然而，还可以使用另一种方法来解决：使用 BSP 树。BSP 可以提供从任何视点观察时，多边形按从前到后和从后到前排列的顺序以及其他有用的信息。

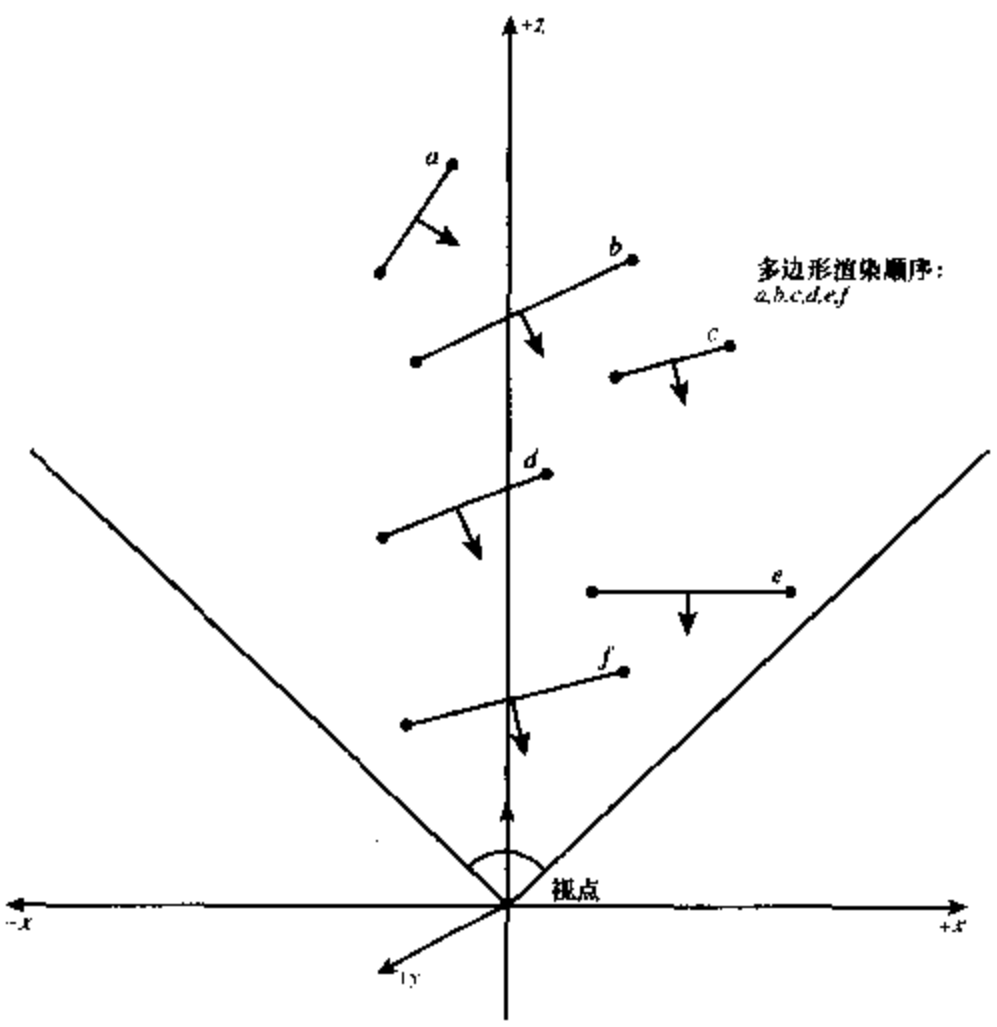


图 13.7　从后到前的渲染顺序

BSP 算法将网格或环境中的所有多边形作为输入，根据分割面将它们划分到凸形子空间中。分割面可以平行于坐标轴、任意的或与多边形共面。下面简要地介绍方法，然后重点讨论最常用的方法。

13.3.1　平行于坐标轴的二元空间划分

图 13.8 描述了一组初始多边形以及使用平行于坐标轴的平面进行空间划分后的结果。正如读者看到的，空间被划分成小型箱体（box），每个箱体中只有一个多边形。如果分割面与多边形相交，通常将多边形分割成两个，并将它们分别加入到分割面对应的两个子空间中。然而，也可以使用另一种方法：在两个子空间（这里为箱体）中都标记该多边形，然后在运行期间将多边形标记为“被处理过”，这样在另一个子空间

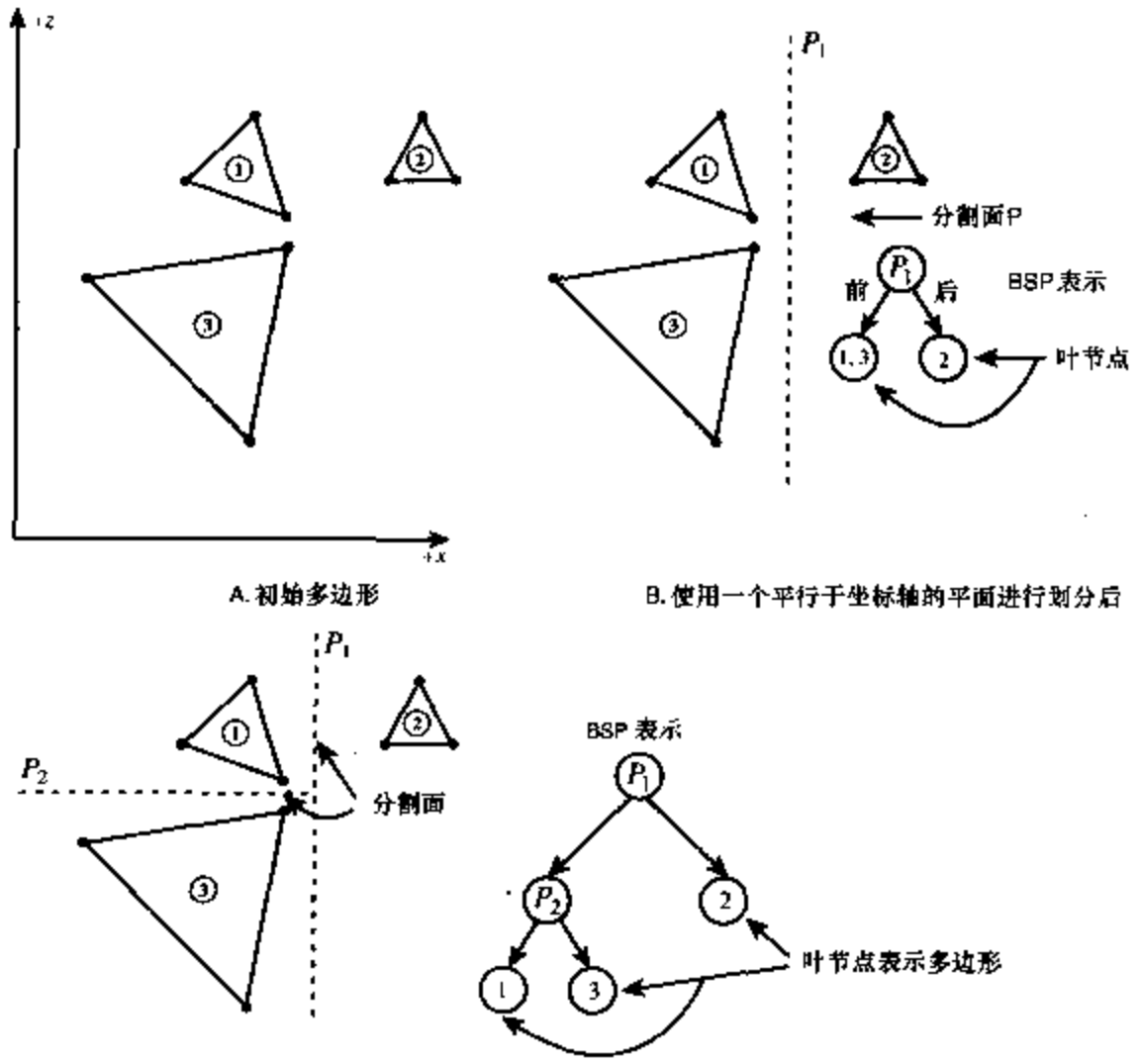


图 13.8　使用平行于坐标轴的分割面进行空间划分

中将忽略该多边形。另外，根据定义，每个子空间都将是凸形的，因为矩形和立方体是凸形的。BSP 的每个节点都定义了一个凸形子空间，稍后将详细讨论这一点。

13.3.2　任意平面空间划分

图 13.9 描述了第二种空间划分方法：选择能够更好地将空间划分的分割面。可能需要使用另一种算法来确定分割面，以便能够将多边形均匀地划分到子空间中或使用最少的分割面。这种空间划分的结果是，生成一个 BSP 树，其中每个节点都定义了一个子空间，而每个叶节点都定义了一个凸形子空间。这样从根节点开始，右子节点定义了分割面后面的所有多边形，左子节点定义了前面的所有多边形。BSP 树的叶节点定义了一个凸形子空间，其中包含一个或多个多边形，如图 13.9 所示。

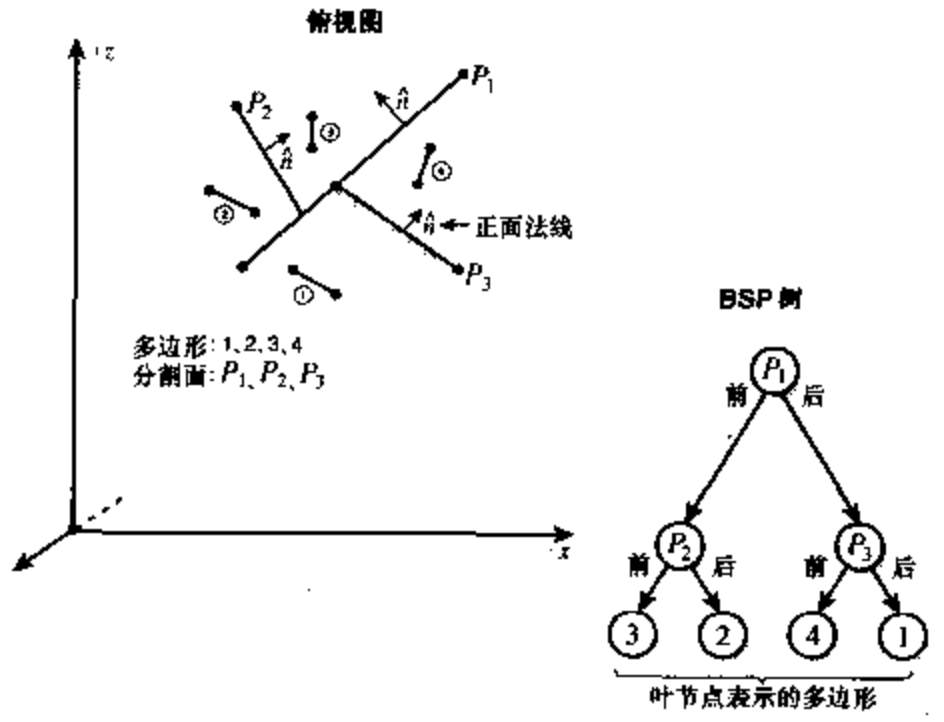


图 13.9　使用任意的分割面

如果读者问，这种划分方法有何帮助呢？则作者的目的便到达了。下面将注意力转移到最后一种空间划分方法。

13.3.3 使用多边形所在的平面来划分空间

这种空间划分方法的工作原理如下：从场景中选择一个多边形，将其用作分割面（就现在而言，选择哪个多边形无关紧要）。多边形所在的平面是无穷大的，被称为超平面（hyperplane）。超平面将空间划分成两个半空间，如图 13.10 所示。

算法根据分割面对场景中的每个多边形进行检测，确定它位于分割面的前面还是后面。如果分割面与多边形相交，则根据交点（两个交点组成一条直线）将其分割成两个多边形，如图 13.11 所示。

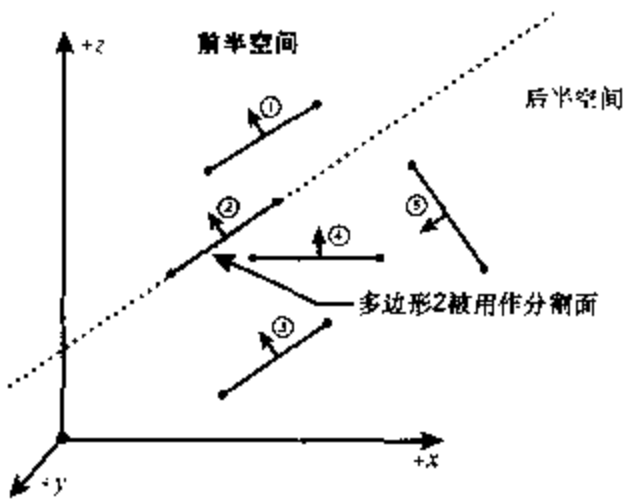


图 13.10 将多边形用作分割面

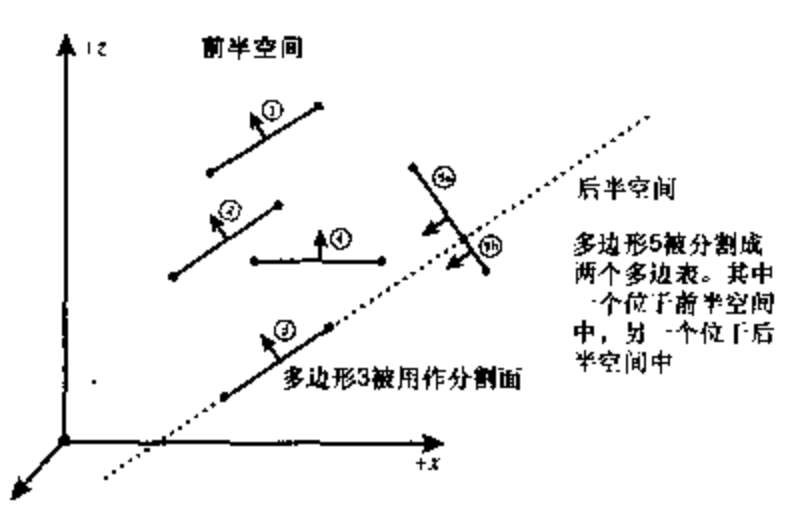


图 13.11 将多边形分割成两个

注意：超平面是一个多维几何学术语，指的是维数比空间本身少 1 的东西。例如，在 3D 空间中，超平面指的是 2D 平面。在 n 维空间中，超平面的维数为 n–1。

算法完成第一步后，递归地对这一步生成每个多边形列表（分别包含位于分割面前面[前半空间]和后面[后半空间]）进行类似地处理。这一过程将不断重复下去，直到每个多边形都被用作分割面，构建好 BSP 树为止。因此，可以将初始多边形链表作为输入提供给空间划分系统。

例如，图 13.12 说明了两个类似的 BSP 树，它们是根据同一个包含 5 个多边形的场景构建而成的，但第一次选择的分割面不同。算法的第一步是创建一个包含分割面的节点，其两个子节点为多边形链表：其中一个包含分割面前面的多边形，另一个包含后面的多边形。然后对这两个多边形链表进行类似的处理：从中选择一个多边形作为分割面（就现在而言，选择哪个都行），然后对多边形进行划分。这一过程不断重复下去，直到得到如图 13.12b 和 13.12c 所示的 BSP 树。

构建的 BSP 树具有一些特殊性质，可利用它们来进行渲染、碰撞检测或大规模剔除。例如，使用一种改进的顺序递归访问算法，可以根据任何视点按从后到前的顺序访问多边形。给定 3D 世界中的任何视点，都可以使用一种简单的顺序搜索算法来确定从后到前渲染多边形的准确顺序（类似于完美的画家算法），该算法的时间复杂度为线性。这样做时，不需要对多边形进行排序，也无需使用 z 缓存。

另外，访问二元查找树（binary search tree，BST）中每个节点的查找时间都是线性的，即复杂度为 O(n)，因为这可以简化为一个线性堆栈问题。查找单个关键字的复杂度为 $O(\log_2^n)$。

注意：选择哪个多边形作为分割面至关重要，因为选择不好会增加划分次数和/或导致非平衡树（前一个原因更为重要），后面将详细讨论这一点。在大多数情况下，可以使用一种简单的试探方法，每次划分时，随机选择 0.1 ~ 1% 的多边形作为候选分割面。然后，根据作为参数传递给优化函数的分割次数和是否要求 BSP 树平衡，选择实际用作分割面的多边形。因此，如果有 10 000 个多边形，每次划分时将最多需要尝试 10 ~ 100 个多边形，才能得到接近于最优的结果。

读者对如何创建 BSP 树有大概了解后，接下来介绍具体的细节，列出编写代码的步骤。假设已经使用某种工具或函数创建了一个多边形链表，其中包含要对其进行空间划分的游戏世界或关卡中的所有多边形，且该列表头名为 root，则创建 BSP 树的算法如下：

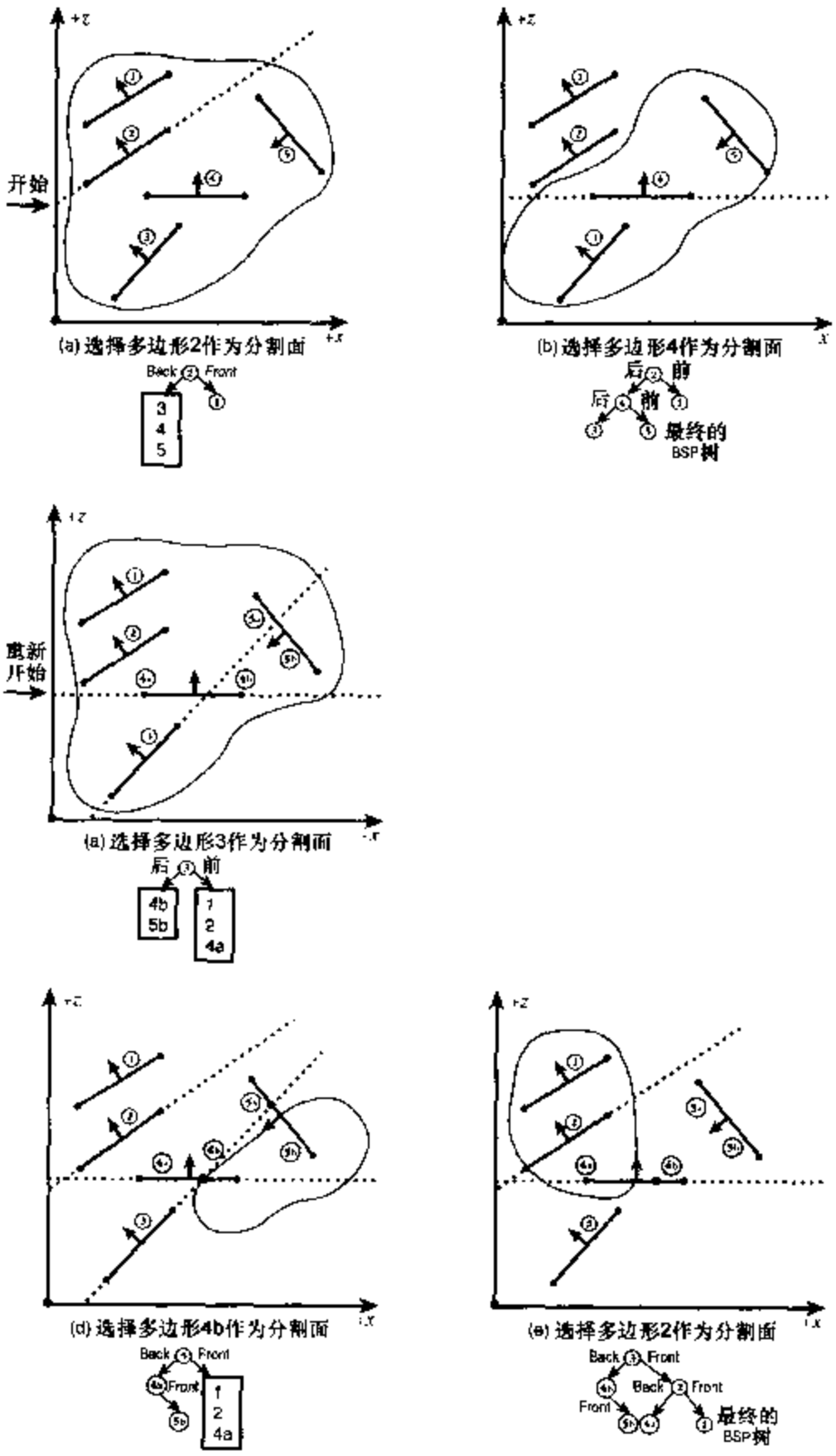


图 13.12　逐步创建 BSP 树

1. 使用某种试探方法从链表中选择一个多边形，将其作为分割面。如果列表中没有其他多边形，则算法结束。

2. 创建两个子列表，它们分别包含位于分割平面前面和后面的多边形，然后将分割平面对应的 BSP 节点的 front 和 back 指针分别指向它们，如图 13.13 所示。

3. 递归处理 front 指针指向的列表。

4. 递归处理 back 指针指向的列表。

当然，需要考虑的细节非常多，如何确定多边形位于分割面的前面还是后面？每次划分时应选择哪个多边形作为分割面？使用什么样的数据结构最合适？

注意：前面已经概述了 BSP 树，现在需要处理具体的细节。我们将实现一个 2D BSP 树。当然，BSP 将被用于 3D 引擎中。然而，作者没有时间来创建功能完备的 3D 建模工具，用于生成 3D BSP；同时计算任意平面与多边形的交点更为复杂，与 BSP 树无关。因此，出于简化的目的，作者将创建由线段组成的 BSP 世界，这些线段将被转换为 3D 墙面。然而，所有的建模和 BSP 树创建工作都将在 2D 空间进行，以降低问题的复杂度。将算法推广到 3D 空间只涉及更多的数学运算，读者应该能够完成这种工作。

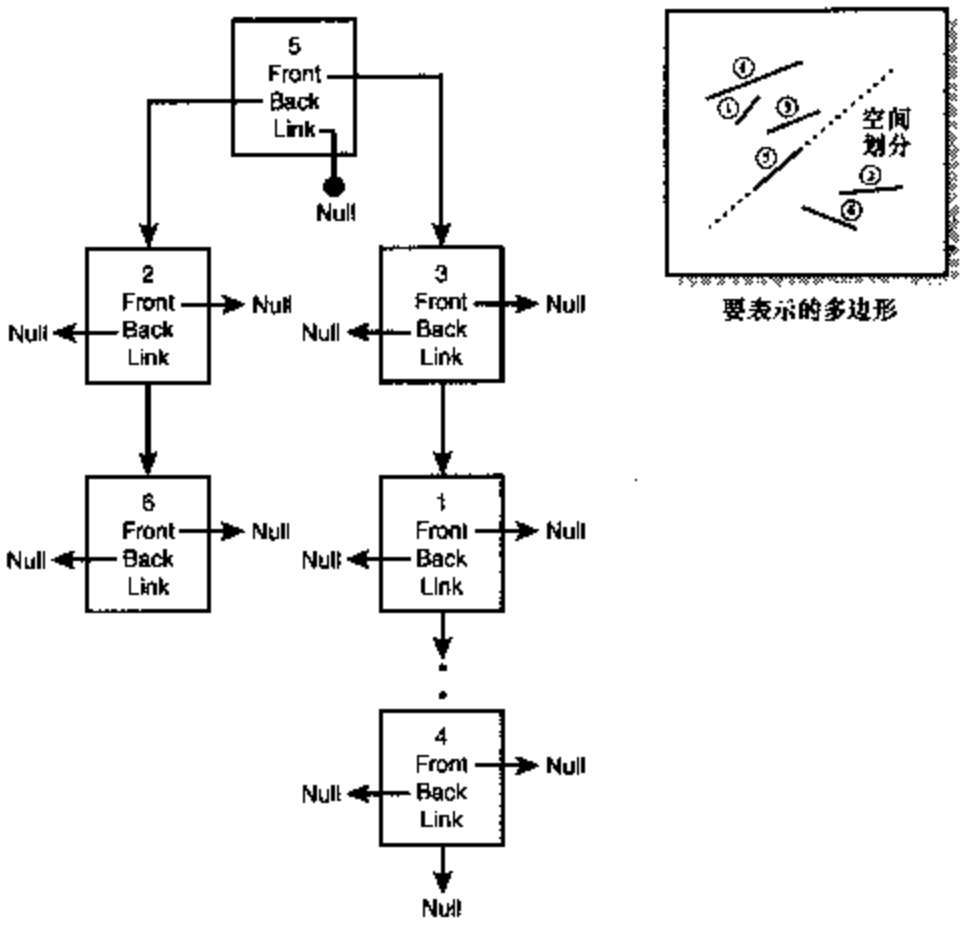


图 13.13　BSP 数据和相关的数据结构

13.3.4　显示/访问 BSP 树中的每个节点

正如前面指出的，BSP 树可用于完成很多工作：渲染、碰撞检测、剔除等。然而，在很多情况下，它被用于根据任意视点位置，确定按从后到前或从前到后渲染游戏关卡中多边形的准确顺序。下面介绍这是如何完成的。

幸运的是，显示 BSP 树比构建它容易得多。只需编写一种这样的算法：它使用改进后的二叉树顺序遍历算法；同时提供一个遍历条件，用于根据观察者位于多边形（分割面）的前面还是后面确定查找路径。

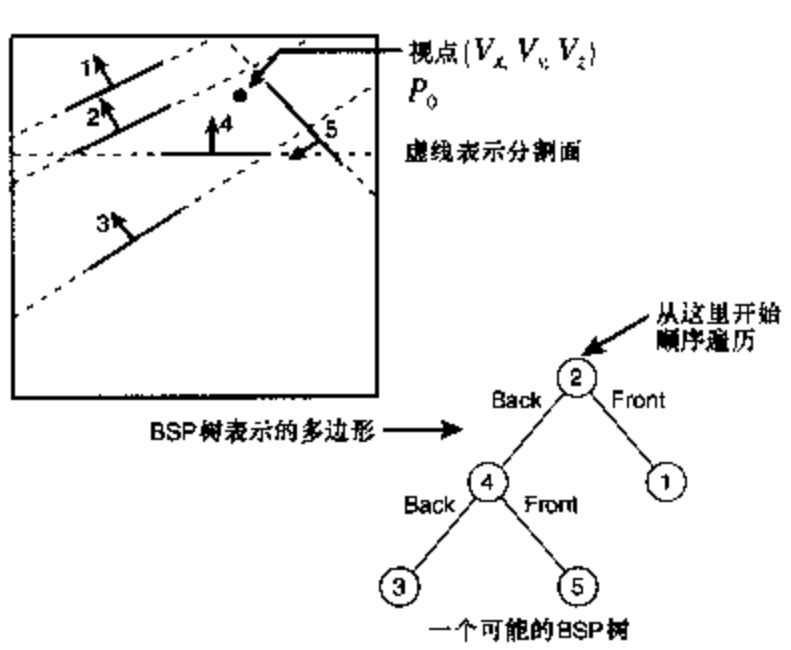


图 13.14　遍历范例中的多边形和 BSP 树

该算法的工作原理如下：如图 13.14 所示，从 BSP 树的根处开始，根据根多边形的平面方程对视点位置进行检测。如果它位于该平面的前面，则首先递归地遍历 back 分支，访问多边形（显示或处理多边形），然后再递归地遍历 front 分支。

相反，如果视点位于分割面的后面，则采取相反的措施：首先递归地遍历 front 分支，访问其中的多边形；然后再递归地遍历 back 分支。

这可能有些令人迷惑，请读者先考虑一会儿，再来看另一个例子。BSP 树的关键之处在于：构建 BSP 树后，不管将视点置于什么地方，视点所在的子空间都是凸形的，且其中包含离视点最近的多边形；该多边形后面的子空间更远。因此，可以利用 BSP 树的这种特性来确定按从后到前渲染多边形的准确顺序。图 13.15 显示了组成 BSP 树的多边形，视点位于 p0 处。

首先，我们发现视点位于多边形 1 的前面，因此多边形 1 是最近的。因为如果视点位于多边形 1 的前面，且在前半空间内没有其他的分割面，根据定义该子空间为凸形的，因此多边形 1 是最近的。根据这种特性可知，位于多边形 1 后面的多边形更远，因此访问这些多边形时，可以递归应用这种思想。所以，如果递归地进行搜索，直到到达离视点最远的多边形，然后回溯并绘制每个节点，将可确保多边形的绘制顺序是从后到前的。同样，如果使用相反的搜索策略访问每个节点，将可确保多边形是按从前到后的顺序正确排列的。

为说明这些概念并加深读者的理解，来看另一个例子。以图 13.14 为例，将上述算法应用于每个多边形，看其是否正确，步骤如下：

1．从多边形 2 开始。视点位于后半空间中，因此首先遍历前半空间。

2．由于多边形 1 没有子节点，因此渲染它，然后回溯（对堆栈执行弹出操作）到多边形 2。

3．渲染多边形 2，然后遍历后半空间。

4．视点位于多边形 4 的前半空间内，因此遍历多边形 4 的后半空间。

5．由于多边形 3 没有子节点，因此渲染它，然后回溯到多边形 4。

6．渲染多边形 4。

7．遍历多边形的前半空间。

8．由于多边形 5 没有子节点，因此渲染它，然后回溯时发现堆栈为空，算法结束。

因此，多边形的绘制顺序为 1、2、3、4 和 5，这是正确的从后到前的顺序。

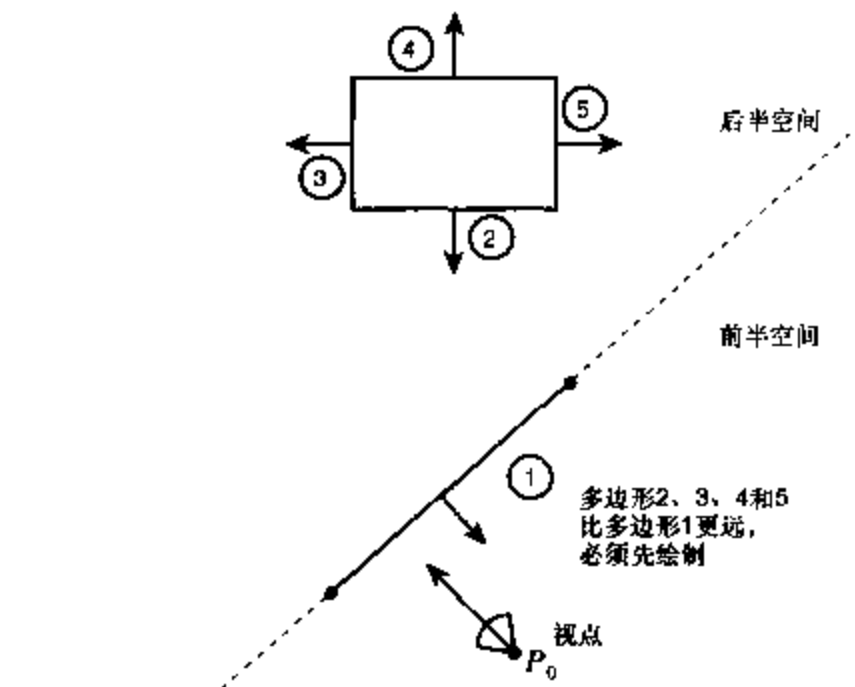


图 13.15　当前子空间中的多边形离视点最近

不管构建 BSP 树时将多边形用作分割面的顺序如何，也不管视点位置和观察方向如何，BSP 树访问算法都是可行的。下面来看一下这种遍历算法的伪代码，假设每个节点都有一个 front 指针和一个 back 指针：

```
void Bsp_Traverse(BSPNODEV1_PTR root, VECTOR4D viewpoint)
{

if (root==NULL)
  return;

if (如果视点位于根多边形的前面)
  {
  // 执行顺序遍历
  Bsp_Traverse(root->back, viewpoint);

  Visit_Polygon(root->poly);

  Bsp_Traverse(root->front, viewpoint);
  }
else
  {
  // 执行顺序遍历
  Bsp_Traverse(root->front, viewpoint);

  Display_Polygon(root->poly);

  Bsp_Traverse(root->back, viewpoint);
  }

  } // end else

} // end Bsp_Traverse
```

非常简单。当然，判断视点位于多边形的前面还是后面需要计算点积。处于简化的目的，这里没有做这样的计算，以便读者能够明白这种遍历算法的优点。

接下来介绍为创建 BSP 演示程序需要提供什么样的数据结构和支持函数，并将它们集成到现有的 3D 引擎中，该引擎支持光照处理、纹理映射和裁剪等。

13.3.5　BSP 树数据结构和支持函数

实现 BSP 树和支持函数没有看起来那么简单——需要做很多决策。即使是创建一个演示这种技术的程

序，也不是一项简单的任务。作者决定使一切尽可能简单，以便读者能够理解其中的思想。首先介绍用表示一堵墙/一个 BSP 节点的数据结构，这将揭示第一个需要解决的问题。

当前，我们的 3D 引擎是完全基于三角形而不是四边形的。做出这样的设计决定旨在简化表示、光照处理、裁剪等工作。然而，建立室内环境模型时，三角形并非最好的选择，因为大多数室内环境都是由平面构成的，这些平面通常是四边形。鉴于这一点，作者决定不使用两个三角形来表示一个四边形，而是创建一个包含 4 个（而不是 3 个）顶点的新多边形数据结构，并使用这种新的表示法来完成与 BSP 相关的工作。然后，在遍历 BSP 树时，可以将每个 BSP 多边形节点转换为两个三角形，如图 13.16 所示。

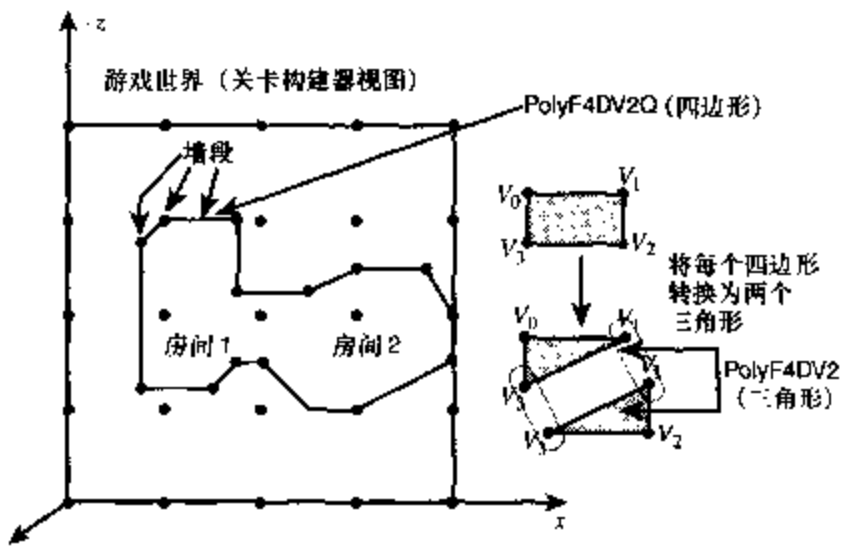


图 13.16　使用四边形表示来简化 BSP 树的创建工作

另一方面，作者想使用一个读者非常熟悉的结构，因此，以 POLYF4DV2 为基础，新增一个顶点，并在名称中加入字母“Q”。下面是用于表示四边形的新多边形数据结构：

```
// 一个自包含的四边形数据结构//////
// 用于表示墙面，因为它们是四边形
typedef struct POLYF4DV2Q_TYP
{
int   state;       // 状态信息
int   attr;       // 多边形的物理属性
int   color;       // 多边形的颜色
int   lit_color[4];  // 用于存储经过光照处理后的颜色
                //使用恒定着色时，颜色存储在元素 0 中
                //使用 Gouraud 着色时，顶点颜色存储在元素 0~3 中
BITMAP_IMAGE_PTR texture; // 指向纹理的指针，用于简单纹理映射

int   mati;     // 材质索引，-1 表示没有材质

float  nlength;    // 多边形法线的长度
VECTOR4D normal;   // 多边形法线

float  avg_z;     // 顶点的平均 z 值，用于 z 排序

VERTEX4DTV1 vlist[4];   // 四边形的顶点
VERTEX4DTV1 tvlist[4];  // 变换后的顶点

POLYF4DV2Q_TYP *next;   // 指向列表中下一个多边形的指针
POLYF4DV2Q_TYP *prev;   // 指向列表中前一个多边形的指针

} POLYF4DV2Q, *POLYF4DV2Q_PTR;s
```

其中与 POLYF4DV2 结构不同的地方为粗体。我们可以使用 POLYF4DV2Q 来表示单个四边形，但还需要将其封装到 BSP 节点中。BSP 节点不仅需要包含一个四边形（表示多边形本身和分割面），还需要包含 front 指针和 back 指针，以便能够构建 BSP 树。用于表示 BSP 节点的数据结构如下：

```
// BSP 节点，所有 BSP 树都是由这种数据结构组成的
typedef struct BSPNODEV1_TYP
{
int    id;   // ID，用于调试
POLYF4DV2Q wall; // 墙面四边形

struct BSPNODEV1_TYP *link; // 指向下一个墙面的指针
struct BSPNODEV1_TYP *front; // 指向 front 子树的指针
struct BSPNODEV1_TYP *back; // 指向 back 子树的指针

} BSPNODEV1, *BSPNODEV1_PTR;
```

BSPNODEV1 是构成 BSP 树的基本单元，图 13.17 对其进行了描述。它由一个四边形、一个用于调试的

数字标识符和 3 个指针组成。其中的两个指针（front 和 back）前面讨论过，最后一个名为 link，用于将所有的 BSP 节点链接成一个链表。二元空间划分算法的输入是通过 link 指针链接的墙面数据，因此任何能够根据游戏关卡的墙面数据创建一个链表的程序都是 BSP 的。该链表将被提供给二元空间划分算法进行处理。

现在，暂时假设已经使用某个程序或工具，通过输入一组墙面数据，生成了一个用 link 指针链接起来的链表，如图 13.18 所示。下面来编写一个函数，它将这种链表作为输入，创建 BSP 树。

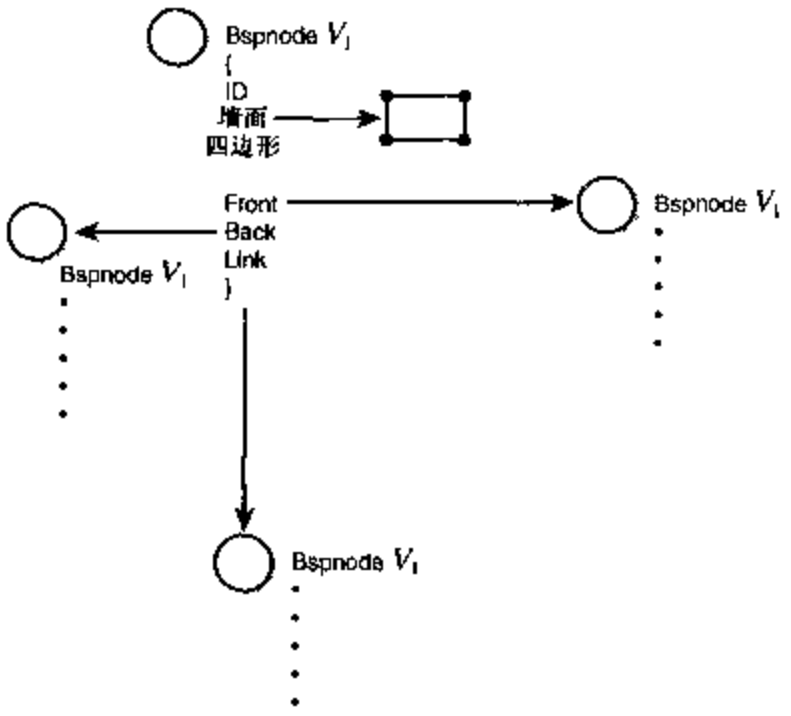


图 13.17　图示 BSPNODEVE1 单元

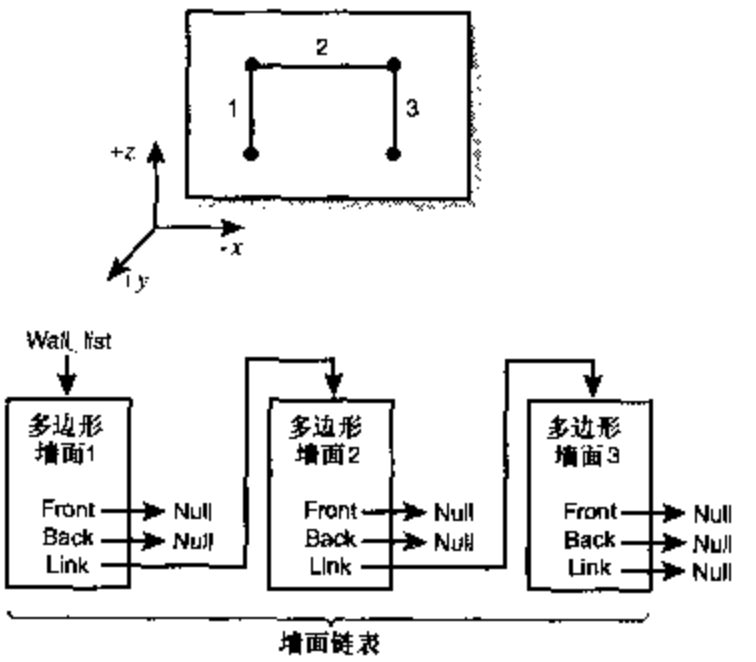


图 13.18　经过处理变成 BSP 树之前的墙面链表

13.3.6　创建 BSP 树

从理论上说，创建 BSP 树并不太难，但涉及指针和内存分配，且共面的墙面有共用的顶点时，问题将变得复杂得多。不过，作者已竭力使函数尽可能清晰易懂。从理论上说，该函数的工作原理如下：它接受墙面链表，将链表中的第一个元素用作分割面，然后将位于该平面前面和后面的墙面分别加入到两个链表中，再将第一个元素的 front 和 back 指针分别指向它们。每个链表是通过墙面数据结构的 link 指针链接起来的。接下来，将每个链表作为输入，递归地调用 BSP 树创建函数。这一过程不断重复下去，直到 front 和 back 指针指向的链表中只有一个多边形或没有多边形位置。至此，BSP 树便创建好了。

在这个处理过程中，有两个主要的难题，同时还有一些特殊情况需要考虑。先来讨论其中的难题。第一个难题是，如何确定多边形位于分割面的前面还是后面。图 13.19 说明了 3 种情形。

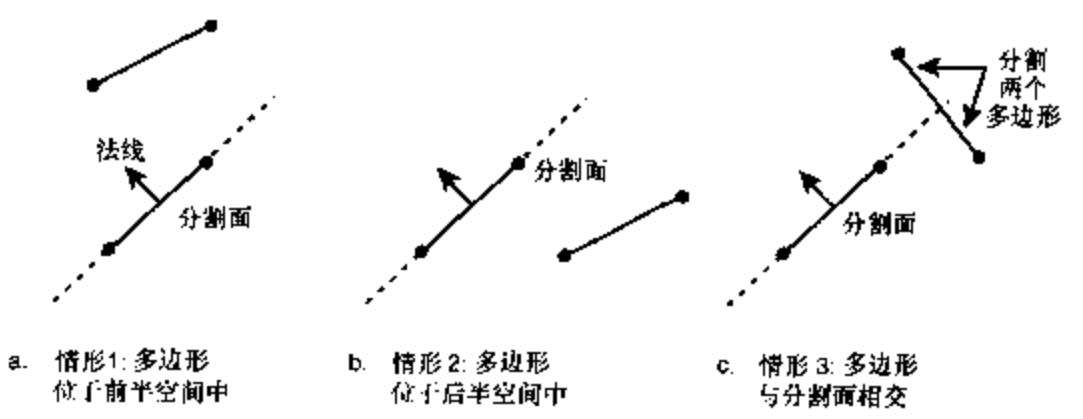


图 13.19　对于每个墙面需要考虑各种情形

情形 1：两个端点都位于分割面的前面。

情形 2：两个端点都位于分割面的后面。

情形 3：一个端点位于分割面的前面，另一个位于分割面的后面，因此需要将多边形分割成两个。

情形 1 和情形 2 检测和处理起来都很容易。在这些情形下，可以根据分割面的面法线和到端点的向量的点积来确定端点位于前面还是后面。位于前面的墙面被加入到 front 链表中，位于后面的多边形被加入到 back 链表中。难以处理的是情形 3。

在这种情况下，需要延长分割面，并计算其与被检测墙前的交点。这很容易，因为墙面被表示为 x-z 面中的 2D 线段，因此计算两个墙面的交点变成了计算两条 2D 直线的交点，如图 13.20 所示。

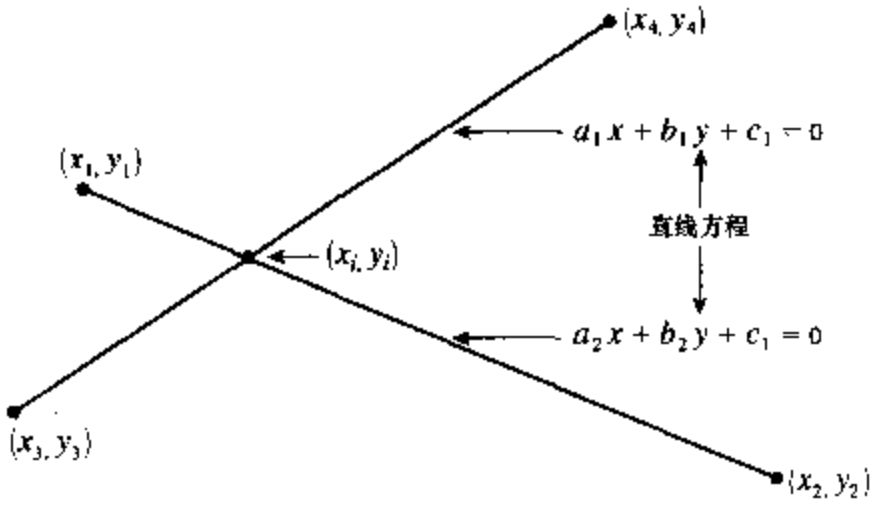


图 13.20　计算两条直线的交点

计算直线交点的方法很多，我们将采用直线方程（而不是参数化方程）的简单方法，因为用参数化形式表示线段需要做额外的工作，而此时这样做在很大程度上说没有任何好处。在实际的 BSP 引擎中，将使用参数化方程，但就现在而言，使用直线方程更容易。另外，创建 BSP 树对时间不敏感（显示它时才对时间敏感），因为将从磁盘中加载 BSP 树，然后使用它，因此增加一些计算量无关紧要。

参见图 13.20 来推导交点的计算公式（读者不应对此感到陌生）。我们需要知道两条直线的方程：

```
a1*x + a1*y = c1
a2*x + a2*y = c2
```

这样，便可以像第 4 章和第 5 章介绍的那样，使用克莱姆法则、矩阵乘法或代入法求解该方程组。作者喜欢使用克莱姆法则，简单地说，该法则指出，可以这样来求解线性方程组：用常量列向量替换系数矩阵中要求解的变量对应的列向量，然后将矩阵视为行列式，并计算它的值；接下来用计算结果去除系数行列式的值，结果便是变量的值。上述方程组的系数矩阵如下：

$$C=\begin{bmatrix} a1 & b1 \\ a2 & b2 \end{bmatrix}$$

因此 Det(C)=(a1*b2 – a2*b1)

要求解 x，需要做如下计算：

$$xi=\frac{\text{行列式}\begin{vmatrix} c1 & b1 \\ c2 & b2 \end{vmatrix}\text{的值}}{(a1*b2-a2*b1)}=\frac{(b2*c1-b1*c2)}{(a1*b2-a2*b1)}$$

使用下述公式，可以计算出 y：

$$yi=\frac{\text{行列式}\begin{vmatrix} a1 & c1 \\ a2 & c2 \end{vmatrix}\text{的值}}{(a1*b2-a2*b1)}=\frac{(a2*c2-a2*c1)}{(a1*b2-a2*b1)}$$

现在惟一的问题是，如何得到直线方程？这很简单，可以使用直线方程的点—斜率形式：

```
m*(x-x0) = (y-y0)
```

其中 m 为斜率，它等于（y1-y0）/（x1-x0）。重新整理上述方程，便可以得到线性形式：

```
(m)*x + (-1)* y = (m*x0-y0)
```

上述方程属于下述形式：

```
a*x + b*y = c
```

其中 a=m，b=-1，c=（m*x0-y0）。

基于上述推导，下面的函数根据两条直线的端点计算它们的交点：

```
void Intersect_Lines(float x0,float y0,float x1,float y1,
           float x2,float y2,float x3,float y3,
           float *xi,float *yi)
{
// 该函数计算并返回两条直线的交点，它假设直线是相交的
// 该函数能够处理垂直直线和水平直线
// 该函数不太巧妙，只是应用数学运算，但由于这是预处理步骤，因此速度无关紧要
```

```
// 如果愿意，也可以使用参数化直线方程，但就这里而言，使用直线方程更简单
// 这里要计算的是两条无穷长直线的交点，而不是线段的交点

float a1,b1,c1, // 直线方程系数常量
   a2,b2,c2,
   det_inv, // 系数矩阵行列式值的倒数
   m1,m2;  // 直线的斜率

// 计算斜率
if ((x1-x0)!=0)
  m1 = (y1-y0) / (x1-x0);
else
  m1 = (float)1.0E+20;  // 表示斜率为无穷大

if ((x3-x2)!=0)
  m2 = (y3-y2) / (x3-x2);
else
  m2 = (float)1.0E+20;  // 表示无穷大斜率

// 计算直线方程系数
a1 = m1;
a2 = m2;

b1 = -1;
b2 = -1;

c1 = (y0-m1*x0);
c2 = (y2-m2*x2);

// 计算行列式值的倒数
det_inv = 1 / (a1*b2 - a2*b1);

// 使用卡莱姆法则来计算 xi 和 yi
*xi=((b1*c2 - b2*c1)*det_inv);
*yi=((a2*c1 - a1*c2)*det_inv);

} // end Intersect_Lines
```

该函数接受两条直线端点的 x、y 坐标和两个用于指向交点坐标的浮点数指针作为参数。

注意：该函数假设两条直线不平行，即相交。如果两条直线是平行的，将发生除零错误，因为两条平行直线的系数行列式的值为 0。这种问题很容易修复，方法是首先判断两条直线是否平行，如果是，则返回一个指出这一点的值。读者调用该函数时，必须确保两条直线不平行。

知道如何计算分割面和墙面的交点后，接下来必须使用交点（xi, yi）将墙面分割成两个墙面，并将其中一个插入到 front 链表中，将另一个插入到 back 链表中。

13.3.7　分割策略

介绍分割策略之前，先来讨论一种优化方法。当墙面与分割面相交时，可以不分割墙面，而是将其原封不动地插入到 front 子树和 back 子树中，同时使用一个字段来记录交点，供以后使用。这样，便无需分割多边形，但有多边形的两个拷贝，它们分别位于 front 子树和 back 子树中。

这看起来好像有问题，但并非如此。遍历 BSP 树时，被访问过的多边形将被标记为访问过。因此，如果多边形有两个拷贝，第一次访问它时将设置被访问标记，这样以后再遇到时将忽略它。另外，多边形可能与多个分割面相交，因此有多个交点。所以，需要使用数组或其他类似的数据结构来记录交点。尽管如此，访问标记算法仍然管用——多边形被访问过后，再次遇到时将忽略它。这将确保遍历 BSP 树时，将按从后到前或从前到后的顺序访问多边形。然而，我们不会这样做，因为它将使本已复杂的工作变得更为复杂。

回到创建 BSP 树的话题，讨论一下特殊情况，这些情况总是会出现。首先，我们假设任何两面墙都不相交，如图 13.21 所示。这不是约束条件，因为要让一堵墙与另一堵墙相交，可以将其表示为两堵墙，交

点两边各一堵墙。

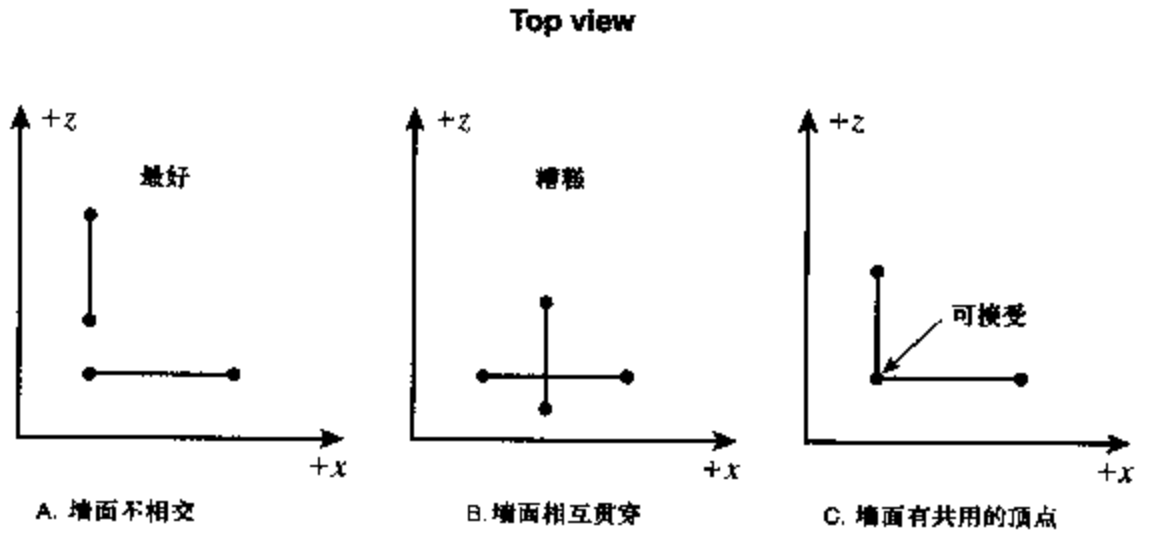


图 13.21　墙面不允许相互贯穿

然而另一种常见的情形——两堵墙有共用的顶点——是不可避免的，必须对其进行处理。请看图 13.22，其中有一组共面的多边形墙段。问题是，在计算期间，不将共用顶点视为位于分割面的前面，也不能将其视为位于分割面的后面。在这种情况下，该如何办呢？

答案是这样问自己：两堵墙有共用边意味着什么？这决定了在这种情况下该如何办。如果两堵墙有共用的垂直边，即在俯视图中，它们有共用的端点，这只意味着共用点既不在分割面的前面，也不在分割面的后面。实际上，它位于分割面上，因此必须对墙面的第二个端点进行检测，来确定墙面位于分割面的前面还是后面。

因此，发现墙面与分割面有共用的边或点后，需要根据墙面的第二个端点或边来确定墙面位于分割面的前面还是后面。然而，还有一个小问题：如果第二个端点或边也位于分割面上，该如何办呢？这不成其为问题——事实上，在这种情况下，将墙面视为位于分割面的前面还是后面无关紧要，因为效果是一样的。

然而，对于共面的墙面，还有另一种优化或简化方法：将它们组合起来。例如，对于图 13.22 所示的墙面，为何分别对各个墙面进行计算呢？我们知道（或通过计算知道），它们是共面的，因此可以使用一条线段或一个墙面来表示它们，这样就根本不会发生墙面共面的情况。

当然，这是以简单性为代价的。表示 BSP 节点的数据结构将更复杂，虽然可以将一系列墙面视为单个墙面，但仍必须能够根据其他分割面来分割它们。因此，必须将它们组合到一个共面墙面链表中，就像它们是一个墙面那样来计算交点，然后就像它们是多个墙面那样分割它们，并将其正确地放置到 front 子树或 back 子树中。图 13.23 说明了这一点。同样，这种优化需要做大量的工作，因此我们不去管它。然而，在实际的引擎中，绝对不要怕麻烦，来编写实现这种功能的代码，因为归根结底，这样做是值得的。

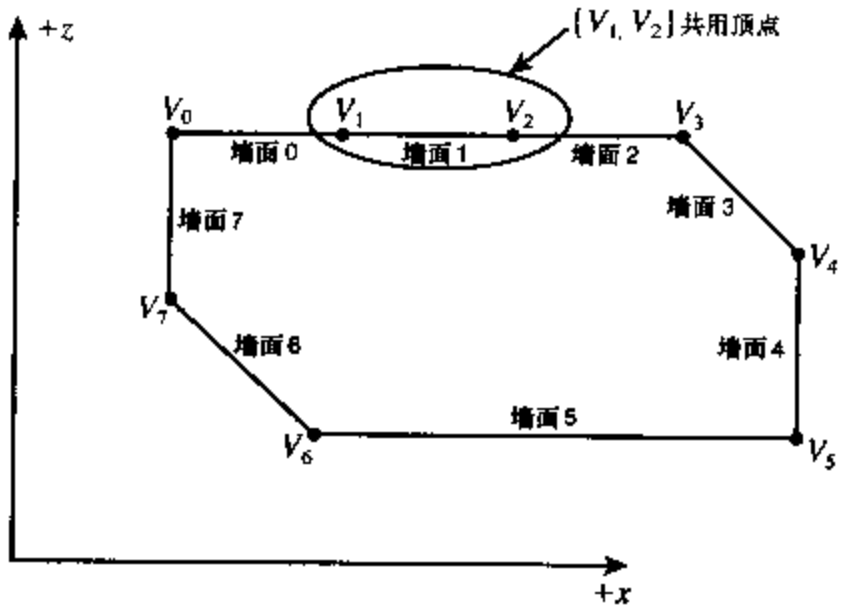


图 13.22　有共用顶点的共面多边形

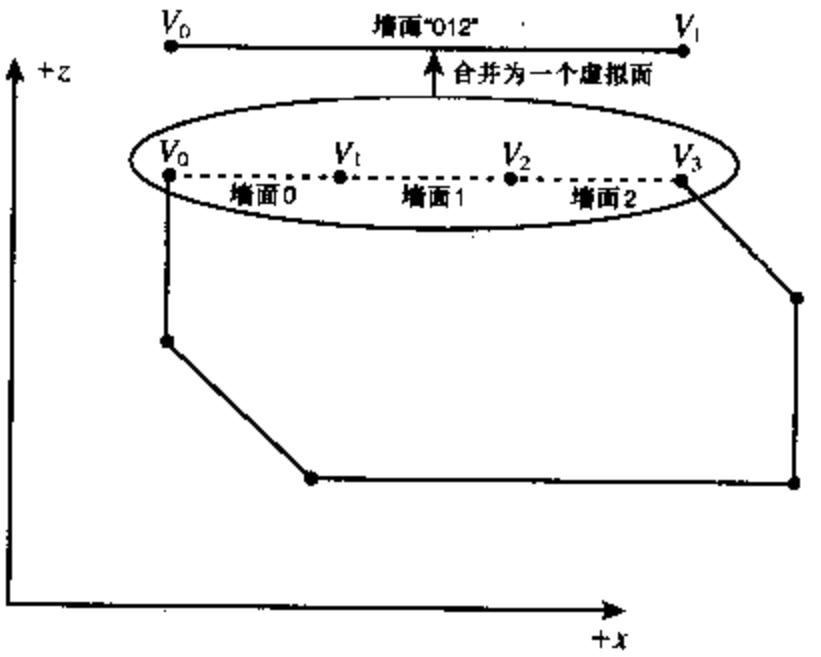


图 13.23　将共面墙面合并成一个面以简化计算

至此，我们从实现的角度概述了这种算法，下面来看一些创建 BSP 树的函数。该函数将以其他方式生成的墙面链表作为参数，其代码如下：

```
void Bsp_Build_Tree(BSPNODEV1_PTR root)
{
// 这个函数根据墙面链表 root 递归地创建一个 BSP 树
// 它将第一个墙面作为分割面，将墙面链表划分成两个墙面列表
// 然后以相同的方式递归地处理这两个链表
// 这种处理结束后，便创建了 BSP 树，其中的每个节点都包含一个墙面

static BSPNODEV1_PTR next_wall,  // 指向要处理的下一个墙面
         front_wall,    // front 墙面
         back_wall,     // back 墙面
         temp_wall;     // 临时墙面

static float dot_wall_1,             // 墙面的点积
       dot_wall_2,
       wall_x0,wall_y0,wall_z0, // 用于测试墙面的工作变量
       wall_x1,wall_y1,wall_z1,
       pp_x0,pp_y0,pp_z0,    //用于分割面的工作变量
       pp_x1,pp_y1,pp_z1,
       xi,zi;           // 分割面和墙面的交点

static VECTOR4D test_vector_1,  // 从分割面到墙面的向量
        test_vector_2;  // 用于判断墙面位于分割面的前面还是后面

static int front_flag = 0,      // 指出墙面位于分割面前面还是后面的标记
      back_flag  = 0,
      index;             // 循环索引

// 第 1 部分 ////////////////////////////////////////////////////////

// 检测 BSP 树是否创建好
if (root==NULL)
  return;

// 将 root 节点包含的墙面用作分隔面
next_wall  = root->link;
root->link = NULL;

// 剔除分割墙面前两个顶点，以简化计算
pp_x0 = root->wall.vlist[0].x;
pp_y0 = root->wall.vlist[0].y;
pp_z0 = root->wall.vlist[0].z;

pp_x1 = root->wall.vlist[1].x;
pp_y1 = root->wall.vlist[1].y;
pp_z1 = root->wall.vlist[1].z;

// 第 2 部分 ////////////////////////////////////////////////////////

// 检测所有墙面是否都已划分
while(next_wall)
   {
   // 检测墙面位于分割面的前面还是后面

   // 首先计算分割面上两个点到墙面上两个点的向量
   VECTOR4D_Build(&root->wall.vlist[0].v,
         &next_wall->wall.vlist[0].v,
         &test_vector_1);

   VECTOR4D_Build(&root->wall.vlist[0].v,
         &next_wall->wall.vlist[1].v,
         &test_vector_2);

   // 计算上述两个向量与墙面法线的点积
   //并分析点积是正还是负来判断墙面上的点位于哪个半空间中
```

```
    dot_wall_1 = VECTOR4D_Dot(&test_vector_1, &root->wall.normal);
    dot_wall_2 = VECTOR4D_Dot(&test_vector_2, &root->wall.normal);

  // 第 3 部分 ////////////////////////////////////////////////////////

    // 进行判断

    // 情形 0: 分割面和墙面有一个共用点
    // 这是一种特殊情况，必须加以考虑

    // 重置标记
    front_flag = back_flag = 0;

    // 判断是否有共用点
    if (VECTOR4D_Equal(&root->wall.vlist[0].v , &next_wall->wall.vlist[0].v) )
     {
     // 分割面的 p0 和墙面的 p0 重叠
     // 因此只需判断墙面的 p1 位于哪边
     if (dot_wall_2 > 0)
       front_flag = 1;
     else
       back_flag = 1;

     } // end if
    else
    if (VECTOR4D_Equal(&root->wall.vlist[0].v, &next_wall->wall.vlist[1].v) )
     {
     // 分割面的 p0 和墙面的 p1 重叠
     // 只需判断墙面的 p0 位于哪边
     if (dot_wall_1 > 0)
       front_flag = 1;
     else
       back_flag = 1;

     } // end if
    else
    if (VECTOR4D_Equal(&root->wall.vlist[1].v,&next_wall->wall.vlist[0].v) )
     {
     // 分割面的 p1 与墙面的 p0 重叠
     // 只需判断墙面的 p1 位于哪边

     if (dot_wall_2 > 0)
       front_flag = 1;
     else
       back_flag = 1;

     } // end if
    else
    if (VECTOR4D_Equal(&root->wall.vlist[1].v, &next_wall->wall.vlist[1].v) )
     {
     // 分割面的 p1 与墙面的 p1 重叠
     // 只需判断墙面的 p0 位于哪边

     if (dot_wall_1 > 0)
       front_flag = 1;
     else
       back_flag = 1;

     } // end if

  // 第 4 部分 ////////////////////////////////////////////////////////

     // 情形 1: 两个点积的符号相同，或者 front_flag 或 back_flag 被设置
     if ( (dot_wall_1 >= 0 && dot_wall_2 >= 0) || front_flag )
       {
       // 将墙面插入到 front 链表中
       if (root->front==NULL)
```

```
        {
        // 这是第一个节点
        root->front     = next_wall;
        next_wall       = next_wall->link;
        front_wall      = root->front;
        front_wall->link = NULL;

        } // end if
      else
        {
        // 这是第 n 个节点
        front_wall->link = next_wall;
        next_wall       = next_wall->link;
        front_wall      = front_wall->link;
        front_wall->link = NULL;

        } // end else

      } // end if both positive

// 第 5 部分 ////////////////////////////////////////////////////////////

    else // 墙面位于分割面后面
    if ( (dot_wall_1 < 0 && dot_wall_2 < 0) || back_flag)
     {
     // 将墙面插入到 back 链表中
     if (root->back==NULL)
      {
      // 这是第一个节点
      root->back    = next_wall;
      next_wall     = next_wall->link;
      back_wall     = root->back;
      back_wall->link = NULL;

      } // end if
     else
       {
       // 这是第 n 个节点
       back_wall->link = next_wall;
       next_wall     = next_wall->link;
       back_wall     = back_wall->link;
       back_wall->link = NULL;

       } // end else

     } // end if both negative

    // 情形 2：两个点积的符号相反，必须将墙面分割成两个

// 第 6 部分 ////////////////////////////////////////////////////////////

    else
    if ( (dot_wall_1 < 0 && dot_wall_2 >= 0) ||
      (dot_wall_1 >= 0 && dot_wall_2 < 0))
     {
     // 分割面与墙面相交，必须将墙面分割成两个

     // 提取墙面的前两个顶点，以简化计算
     wall_x0 = next_wall->wall.vlist[0].x;
     wall_y0 = next_wall->wall.vlist[0].y;
     wall_z0 = next_wall->wall.vlist[0].z;

     wall_x1 = next_wall->wall.vlist[1].x;
     wall_y1 = next_wall->wall.vlist[1].y;
     wall_z1 = next_wall->wall.vlist[1].z;

     // 计算墙面和分割面的交点
```

```
// 交点是在 x-z 平面中计算的
Intersect_Lines(wall_x0, wall_z0, wall_x1, wall_z1,
        pp_x0,   pp_z0,   pp_x1,   pp_z1,
       &xi, &zi);

// 需要分割墙面，得到两个新的墙面
// 然后将这两个墙面加入到 front 或 back 链表中，并删除原来的墙面

// 处理第一个墙面...

// 为墙面分配内存
temp_wall = (BSPNODEV1_PTR)malloc(sizeof(BSPNODEV1));

// 设置指针
temp_wall->front = NULL;
temp_wall->back  = NULL;
temp_wall->link  = NULL;

// 多边形法线不变
temp_wall->wall.normal  = next_wall->wall.normal;
temp_wall->wall.nlength = next_wall->wall.nlength;

// 多边形颜色不变
temp_wall->wall.color = next_wall->wall.color;

// 材质不变
temp_wall->wall.mati = next_wall->wall.mati;

// 纹理不变
temp_wall->wall.texture = next_wall->wall.texture;

// 属性不变
temp_wall->wall.attr = next_wall->wall.attr;

// 状态不变
temp_wall->wall.state = next_wall->wall.state;

temp_wall->id  = next_wall->id + WALL_SPLIT_ID; // 修改 ID
// 计算顶点坐标
for (index = 0; index < 4; index++)
 {
  temp_wall->wall.vlist[index].x = next_wall->wall.vlist[index].x;
  temp_wall->wall.vlist[index].y = next_wall->wall.vlist[index].y;
  temp_wall->wall.vlist[index].z = next_wall->wall.vlist[index].z;
  temp_wall->wall.vlist[index].w = 1;

 // 复制顶点属性、纹理坐标和法线
 temp_wall->wall.vlist[index].attr = next_wall->wall.vlist[index].attr;
 temp_wall->wall.vlist[index].n    = next_wall->wall.vlist[index].n;
 temp_wall->wall.vlist[index].t    = next_wall->wall.vlist[index].t;
 } // end for index

// 将顶点 1 和 2 的坐标设置为交点的坐标
// 但坐标 y 不变
temp_wall->wall.vlist[1].x = xi;
temp_wall->wall.vlist[1].z = zi;

temp_wall->wall.vlist[2].x = xi;
temp_wall->wall.vlist[2].z = zi;

// 第 7 部分 ////////////////////////////////////////////////////////

// 将新的墙面插入到 front 或 back 链表中
if (dot_wall_1 >= 0)
  {
  // 将新墙面插入到 front 链表中
  if (root->front==NULL)
```

```
    {
    // 这是第一个节点
    root->front   = temp_wall;
    front_wall    = root->front;
    front_wall->link = NULL;

    } // end if
   else
    {
    // 这是第 n 个节点
    front_wall->link = temp_wall;
    front_wall    = front_wall->link;
    front_wall->link = NULL;

    } // end else

  } // end if positive
else
if (dot_wall_1 < 0)
  {
  // 将新墙面插入到 back 链表中
  if (root->back==NULL)
    {
    // 这是第一个节点
    root->back    = temp_wall;
    back_wall     = root->back;
    back_wall->link = NULL;

    } // end if
  else
    {
    // 这是第 n 个节点
    back_wall->link = temp_wall;
    back_wall     = back_wall->link;
    back_wall->link = NULL;

    } // end else

  } // end if negative

// 第 8 部分 ////////////////////////////////////////////////////////////////

  // 处理第二个新墙面...

  // 为新墙面分配内存
  temp_wall = (BSPNODEV1_PTR)malloc(sizeof(BSPNODEV1));

  // 设置指针
  temp_wall->front = NULL;
  temp_wall->back  = NULL;
  temp_wall->link  = NULL;

  // 面法线不变
  temp_wall->wall.normal  = next_wall->wall.normal;
  temp_wall->wall.nlength = next_wall->wall.nlength;

  // 颜色不变
  temp_wall->wall.color = next_wall->wall.color;

  // 材质不变
  temp_wall->wall.mati = next_wall->wall.mati;

  // 纹理不变
  temp_wall->wall.texture = next_wall->wall.texture;

  // 属性不变
  temp_wall->wall.attr = next_wall->wall.attr;
```

```
// 状态不变
temp_wall->wall.state = next_wall->wall.state;

temp_wall->id  = next_wall->id + WALL_SPLIT_ID; // 修改 ID
// 计算新墙面的顶点坐标
for (index=0; index < 4; index++)
 {
 temp_wall->wall.vlist[index].x = next_wall->wall.vlist[index].x;
 temp_wall->wall.vlist[index].y = next_wall->wall.vlist[index].y;
 temp_wall->wall.vlist[index].z = next_wall->wall.vlist[index].z;
 temp_wall->wall.vlist[index].w = 1;

 // 复制顶点属性、纹理坐标和法线
 temp_wall->wall.vlist[index].attr = next_wall->wall.vlist[index].attr;
 temp_wall->wall.vlist[index].n    = next_wall->wall.vlist[index].n;
 temp_wall->wall.vlist[index].t    = next_wall->wall.vlist[index].t;

 } // end for index

// 将顶点的坐标 0 和 3 设置为交点的坐标
// 但坐标 y 保持不变
temp_wall->wall.vlist[0].x = xi;
temp_wall->wall.vlist[0].z = zi;

temp_wall->wall.vlist[3].x = xi;
temp_wall->wall.vlist[3].z = zi;

// 将新墙面插入到 front 或 back 链表中
if (dot_wall_2 >= 0)
  {
  // 将墙面插入到 front 链表中
  if (root->front==NULL)
   {
   // 这是第一个节点
   root->front   = temp_wall;
   front_wall    = root->front;
   front_wall->link = NULL;

   } // end if
  else
   {
   // 这是第 n 个节点
   front_wall->link = temp_wall;
   front_wall    = front_wall->link;
   front_wall->link = NULL;

   } // end else


  } // end if positive
else
if (dot_wall_2 < 0)
  {
  // 将墙面插入到 back 链表中
  if (root->back==NULL)
   {
   // 这是第一个节点
   root->back   = temp_wall;
   back_wall    = root->back;
   back_wall->link = NULL;

   } // end if
  else
   {
   // 这是第 n 个节点
   back_wall->link = temp_wall;
```

```
        back_wall    = back_wall->link;
        back_wall->link = NULL;

        } // end else

       } // end if negative

// 第 9 部分 ////////////////////////////////////////////////////////

      // 将原来的墙面删除
      temp_wall = next_wall;
      next_wall = next_wall->link;

      // 释放内存
      free(temp_wall);

      } // end else

    } // end while

// 第 10 部分 ///////////////////////////////////////////////////////

// 递归地处理 front 和 back 链表
Bsp_Build_Tree(root->front);

Bsp_Build_Tree(root->back);

} // end Bsp_Build_Tree
```

作者将这个函数分成了多个部分，各个部分的功能如下：

- **第 1 部分**：检测当前的 BSPNODEV1 是否为 NULL。如果不是，则提取分割面的顶点。
- **第 2 部分**：根据墙面链表中第一个墙面的两个顶点，使用点积来确定该墙面相对于分割面的位置。
- **第 3 部分**：处理共用端点的特殊情况，并设置相应的标记。
- **第 4 部分**：根据点积的符号确定墙面是否在分割面的前面，如果是，将其加入到 front 链表中。
- **第 5 部分**：根据点积的符号确定墙面是否在分割面的后面，如果是，将其加入到 back 链表中。
- **第 6 部分**：墙面被分割面划分成两部分，因此计算新端点的坐标。
- **第 7 部分**：将分割得到的第一个墙面插入到 front 或 back 链表中。
- **第 8 部分**：将分割得到的第二个墙面插入到 front 或 back 链表中。
- **第 9 部分**：提取并处理下一个墙面。
- **第 10 部分**：处理完墙面链表中所有的墙面后，递归地处理 front 链表和 back 链表。

该函数退出时，BSP 树便创建好了，原来的墙面链表被破坏：内存中的内容不变，但不能再使用 link 指针来遍历墙面链表了。有了 BSP 树后，来看如何显示它。

13.3.8　遍历和显示 BSP 树

前面介绍过遍历 BSP 树的算法，但如何在图形流水线中实现它呢？答案是，将遍历 BSP 树，并在遍历期间将被访问的多边形插入到渲染列表中，然后像处理其他物体一样，将渲染列表传递给流水线中其他的模块。当然，遍历 BSP 树并将每个多边形插入到渲染列表中时，必须将每个四边形分割成两个三角形，并计算三角形的顶点坐标、纹理坐标等，如图 13.24 所示。

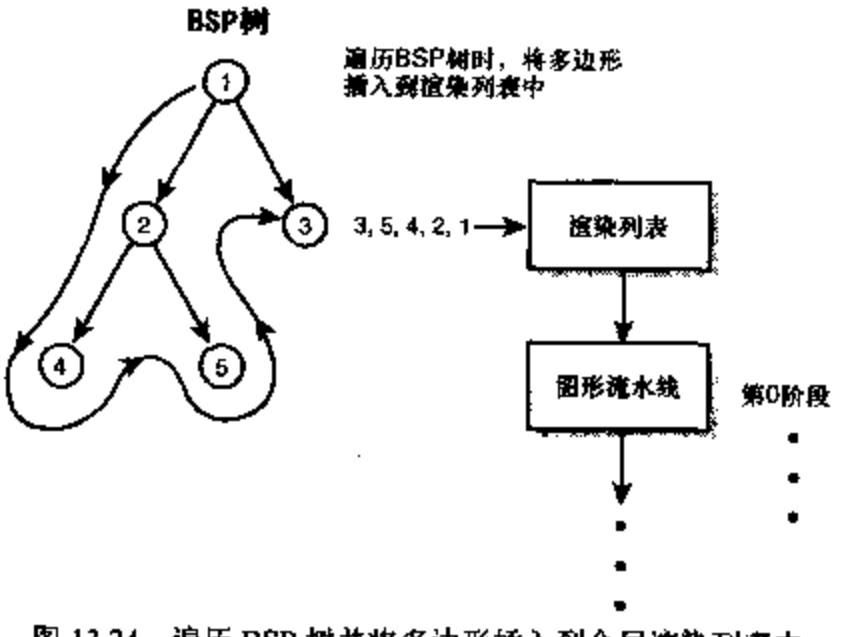


图 13.24　遍历 BSP 树并将多边形插入到全局渲染列表中

这种方法的优点是，无需对多边形进行排序，也不用在渲染期间使用 z 缓存；而只需使用改进的顺序遍历算法来插入 BSP 树中的多边形，使其按从

后到前的顺序被插入到渲染列表中（类似于画家算法）。多边形的排列顺序将是正确的，因为将在以改进的顺序遍历算法递归地遍历 BSP 树时将多边形插入到渲染列表中，这种方法可确保对于任何视点，多边形的排列顺序都是正确的。下面是遍历 BSP 树并将其中的多边形插入到渲染列表中的函数：

```
void Bsp_Insertion_Traversal_RENDERLIST4DV2(RENDERLIST4DV2_PTR rend_list,
   BSPNODEV1_PTR root,
   CAM4DV1_PTR cam,
   int insert_local=0)

{
// 将 BSP 树中的节点转换为多边形面
// 并将可见、处于活动状态、未被剔除和裁剪掉的多边形插入到渲染列表中
// 参数 insert_local 指定使用顶点列表 vlist_local 还是 vlist_trans
// 如果该参数为 1，则插入 BSP 树中未经变换的多边形
// 默认值为 0，即只插入至少经过了局部坐标到世界坐标变换的多边形

// 该函数根据参数 cam 中指定的视点，按从后到前的顺序递归遍历 BSP 树
// BSP 树中多边形的坐标必须是世界坐标

// 该函数检测视点位于当前墙面的前面还是后面，然后执行相应的处理
// 并将每个多边形插入到渲染列表中

static VECTOR4D test_vector;
static float dot_wall;

// 第 1 部分 ////////////////////////////////////////////////////////////

//Write_Error("\nEntering Bsp_Insertion_Traversal_RENDERLIST4DV2()...");

//Write_Error("\nTesting root...");

// BSP 树是否为空?
if (root==NULL)
  {
  //Write_Error("\nRoot was null...");
  return;
  } // end if

//Write_Error("\nRoot was valid...");

// 检测视点位于当前墙面的前面还是后面
VECTOR4D_Build(&root->wall.vlist[0].v, &cam->pos, &test_vector);

// 计算测试向量和墙面向量的点积
dot_wall = VECTOR4D_Dot(&test_vector, &root->wall.normal);

//Write_Error("\nTesting dot product...");

// 第 2 部分 ////////////////////////////////////////////////////////////

// 如果点积大于 0，则视点位于当前墙面的前面
// 因此递归地处理 back 子树，然后再递归地处理 front 子树
// 否则，执行相反的处理
if (dot_wall > 0)
  {
  // 视点位于当前墙面的前面
  //Write_Error("\nDot > 0, front side...");

  // 处理 back 子树
  Bsp_Insertion_Traversal_RENDERLIST4DV2(rend_list,
                      root->back,
                      cam, insert_local);

  // 将四边形分割成两个三角形
  POLYF4DV2 poly1, poly2;
```

```
// POLYF4DV2 和 POLYF4DV2Q 之间的惟一差别在于：
// 后者有 4 个顶点而不是 3 个，因此需要将其分割成两个三角形
// 复制重要的字段
poly1.state   = root->wall.state;      // 状态信息
poly1.attr    = root->wall.attr;       // 物理属性
poly1.color   = root->wall.color;      // 颜色
poly1.texture = root->wall.texture;    // 指向纹理的指针，用于简单纹理映射
poly1.mati    = root->wall.mati;       // 材质索引，-1 表示没有材质
poly1.nlength = root->wall.nlength;    // 多边形法线的长度
poly1.normal  = root->wall.normal;     // 多边形法线

poly2.state   = root->wall.state;      // 状态信息
poly2.attr    = root->wall.attr;       // 物理属性
poly2.color   = root->wall.color;      // 颜色
poly2.texture = root->wall.texture;    // 指向纹理的指针，用于简单纹理映射
poly2.mati    = root->wall.mati;       // 材质索引，-1 表示没有材质
poly2.nlength = root->wall.nlength;    // 多边形法线的长度
poly2.normal  = root->wall.normal;     // 多边形法线

// 四边形的顶点如下
// v0      v1
//
//
// v3      v2
// 我们需要创建两个这样的三角形
//    三角形 1      三角形 2
// v0      v1        v1
//
//
//
// v3        v3     v2
//
// 其中三角形 1 的顺时针环绕顺序为 v0、v1、v3
// 三角形 2 的顺时针环绕顺序为 v1、v2、v3
if (insert_local==1)
 {
 // 设置三角形 1 的顶点坐标
 poly1.vlist[0]  = root->wall.vlist[0];
 poly1.tvlist[0] = root->wall.vlist[0];

 poly1.vlist[1]  = root->wall.vlist[1];
 poly1.tvlist[1] = root->wall.vlist[1];

 poly1.vlist[2]  = root->wall.vlist[3];
 poly1.tvlist[2] = root->wall.vlist[3];

 // 设置三角形 2 的顶点坐标
 poly2.vlist[0]  = root->wall.vlist[1];
 poly2.tvlist[0] = root->wall.vlist[1];

 poly2.vlist[1]  = root->wall.vlist[2];
 poly2.tvlist[1] = root->wall.vlist[2];

 poly2.vlist[2]  = root->wall.vlist[3];
 poly2.tvlist[2] = root->wall.vlist[3];
 } // end if
else
 {
 // 设置三角形 1 的顶点坐标
 poly1.vlist[0]  = root->wall.vlist[0];
 poly1.tvlist[0] = root->wall.tvlist[0];

 poly1.vlist[1]  = root->wall.vlist[1];
 poly1.tvlist[1] = root->wall.tvlist[1];

 poly1.vlist[2]  = root->wall.vlist[3];
 poly1.tvlist[2] = root->wall.tvlist[3];
```

```
    // 设置三角形 2 的顶点坐标
    poly2.vlist[0]  = root->wall.vlist[1];
    poly2.tvlist[0] = root->wall.tvlist[1];

    poly2.vlist[1]  = root->wall.vlist[2];
    poly2.tvlist[1] = root->wall.tvlist[2];

    poly2.vlist[2]  = root->wall.vlist[3];
    poly2.tvlist[2] = root->wall.tvlist[3];
    } // end if

    //Write_Error("\nInserting polygons...");

    // 将多边形插入到渲染列表中
    Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly1);
    Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly2);

    // 递归处理 front 子树
    Bsp_Insertion_Traversal_RENDERLIST4DV2(rend_list, root->front,
                        cam, insert_local);

    } // end if

// 第 3 部分 ////////////////////////////////////////////////////////

else
  {
  // 视点位于当前墙面的后面
  //Write_Error("\nDot < 0, back side...");

  // 处理 back 子树
  Bsp_Insertion_Traversal_RENDERLIST4DV2(rend_list, root->front,
                      cam, insert_local);

  // 将四边形分割成两个三角形，以便插入到渲染列表中
  POLYF4DV2 poly1, poly2;

  // POLYF4DV2 和 POLYF4DV2Q 之间的惟一差别在于：
  // 后者有 4 个顶点而不是 3 个，因此需要将其分割成两个三角形
  // 复制重要的字段
  poly1.state   = root->wall.state;      // 状态信息
  poly1.attr    = root->wall.attr;       // 物理属性
  poly1.color   = root->wall.color;      // 颜色
  poly1.texture = root->wall.texture;    // 指向纹理的指针，用于简单纹理映射
  poly1.mati    = root->wall.mati;       // 材质索引，-1 表示没有材质
  poly1.nlength = root->wall.nlength;    // 多边形法线的长度
  poly1.normal  = root->wall.normal;     // 多边形法线

  poly2.state   = root->wall.state;      // 状态信息
  poly2.attr    = root->wall.attr;       // 物理属性
  poly2.color   = root->wall.color;      // 颜色
  poly2.texture = root->wall.texture;    // 指向纹理的指针，用于简单纹理映射
  poly2.mati    = root->wall.mati;       // 材质索引，-1 表示没有材质
  poly2.nlength = root->wall.nlength;    // 多边形法线长度
  poly2.normal  = root->wall.normal;     // 多边形法线

  // 四边形的顶点如下
  // v0      v1
  //
  //
  // v3      v2
  // 我们需要创建两个这样的三角形
  //    三角形 1              三角形 2
  // v0      v1                 v1
  //
  //
```

```
//
// v3              v3      v2
//
// 其中三角形 1 的顺时针环绕顺序为 v0、v1、v3
// 三角形 2 的顺时针环绕顺序为 v1、v2、v3
if (insert_local==1)
 {
 // 设置三角形 1 的顶点坐标
 poly1.vlist[0]  = root->wall.vlist[0];
 poly1.tvlist[0] = root->wall.vlist[0];

 poly1.vlist[1]  = root->wall.vlist[1];
 poly1.tvlist[1] = root->wall.vlist[1];

 poly1.vlist[2]  = root->wall.vlist[3];
 poly1.tvlist[2] = root->wall.vlist[3];

 // 设置三角形 2 的顶点坐标
 poly2.vlist[0]  = root->wall.vlist[1];
 poly2.tvlist[0] = root->wall.vlist[1];

 poly2.vlist[1]  = root->wall.vlist[2];
 poly2.tvlist[1] = root->wall.vlist[2];

 poly2.vlist[2]  = root->wall.vlist[3];
 poly2.tvlist[2] = root->wall.vlist[3];
 } // end if
else
 {
 // 设置三角形 1 的顶点坐标
 poly1.vlist[0]  = root->wall.vlist[0];
 poly1.tvlist[0] = root->wall.tvlist[0];

 poly1.vlist[1]  = root->wall.vlist[1];
 poly1.tvlist[1] = root->wall.tvlist[1];

 poly1.vlist[2]  = root->wall.vlist[3];
 poly1.tvlist[2] = root->wall.tvlist[3];

 // 设置三角形 2 的顶点坐标
 poly2.vlist[0]  = root->wall.vlist[1];
 poly2.tvlist[0] = root->wall.tvlist[1];

 poly2.vlist[1]  = root->wall.vlist[2];
 poly2.tvlist[1] = root->wall.tvlist[2];

 poly2.vlist[2]  = root->wall.vlist[3];
 poly2.tvlist[2] = root->wall.tvlist[3];
 } // end if

//Write_Error("\nInserting polygons...");

// 将多边形插入到渲染列表中
Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly1);
Insert_POLYF4DV2_RENDERLIST4DV2(rend_list, &poly2);

// 递归地处理 back 子树
Bsp_Insertion_Traversal_RENDERLIST4DV2(rend_list, root->back,
                    cam, insert_local);

} // end else

//Write_Error("\nExiting Bsp_Insertion_Traversal_RENDERLIST4DV2()...");

} // end Bsp_Insertion_Traversal_RENDERLIST4DV2
```

和以往一样，函数的实际实现版本与伪代码有天壤之别。正如读者看到的，上述函数比前面介绍的伪

代码要复杂得多。下面来看看各个部分的功能：

- **第 1 部分**——检测当前节点是否为 NULL。如果是，则退出；否则计算点积。
- **第 2 部分**——根据点积的符号确定视点（观察者）位于前面还是后面，并据此从两种不同的遍历方法中选择一种。这两种方法处理递归调用不同外，其他内容相同。
- **第 3 部分**——首先通过递归调用处理后面的墙面；然后将多边形分割成两个三角形，复制顶点坐标、纹理坐标和其他信息，并将这两个三角形加入到渲染列表中；最后再通过递归调用来处理后面的墙面。

第 2 部分和第 3 部分相同，只是递归处理 back 和 front 子树的顺序相反。

调用上述函数后，BSP 树中的多边形被插入到渲染列表中；接下来只需像将物体插入到渲染列表中后那样使用渲染流水线即可。

然而，还有两个问题需要解决。首先，读者可能注意到了，该函数有一个名为 insert_local 的参数，它指定将墙面多边形插入到渲染列表中时，是使用局部顶点列表 vlist[ ]还是变换后的顶点列表 tvlist[ ]。BSP 树位于局部空间中，但由于它是一个游戏关卡的网格，局部空间和世界空间可能相同，因此这两个数组中包含的数据可能相同。然而，如果不是这样，且要对 BSP 树进行变换，则只需（以任何顺序）遍历它，并对每个节点进行变换。例如，要对 BSP 树中的每个顶点进行平移，可以编写这样的函数：

```
void Bsp_Translate(BSPNODEV1_PTR root, VECTOR4D_PTR trans)
{
// 这个函数对 BSP 树中的所有墙面进行平移
// 它是递归的，我们并不需要这个函数
// 但它很好地说明了如何以递归方式对 BSP 树和类似的树形结构进行变换
// 这里对局部坐标进行平移，将结果存储到变换后的顶点列表中

static int index; // 循环变量

// 检测 BSP 树是否为空
if (root==NULL)
  return;

// 对 back 子树进行平移
Bsp_Translate(root->back, trans);

// 遍历当前墙面的所有顶点，并对其进行平移
for (index=0; index < 4; index++)
  {
  // 执行平移
  root->wall.tvlist[index].x = root->wall.vlist[index].x + trans->x;
  root->wall.tvlist[index].y = root->wall.vlist[index].y + trans->y;
  root->wall.tvlist[index].z = root->wall.vlist[index].x + trans->z;
  } // end for index

// 对 front 子树进行平移
Bsp_Translate(root->front, trans);

} // end Bsp_Translate
```

然后将 BSP 树的根和一个平移向量作为参数来调用上述函数，这样整个 BSP 树将被平移。

注意：读者可能对递归的效率表示怀疑。作者尝试过将很多函数转换为纯粹的基于堆栈的数据递归，发现编译器的效率与作者所做的一样高。最终，作者发现，如果函数包含函数体，通过堆栈将其转换为数据递归以节省函数调用开销几乎毫无意义。

一个更通用的变换函数是，遍历 BSP 树，并使用一个矩阵来变换所有的顶点。另外，和很多变换函数一样，可以选择对哪种坐标进行变换：对局部坐标进行变换、对变换后的坐标进行变换、对局部坐标进行变换并将结果作为变换后的坐标。下面的函数实现了这些功能：

```
void Bsp_Transform(BSPNODEV1_PTR root,  // BSP 树根
          MATRIX4X4_PTR mt,  // 变换矩阵
          int coord_select)  // 选择变换的坐标
{
```

```
// 这个函数遍历 BSP 树，并将变换矩阵应用于每个节点
// 它是递归的，使用顺序遍历
// 当然也可以使用先根顺序（preorder）或后根顺序遍历

// 检测是否达到叶节点
if (root==NULL)
  return;

// 对 back 子树进行变换
Bsp_Transform(root->back, mt, coord_select);

// 遍历当前墙面的所有顶点，并将其变换为相机坐标

// 对哪种坐标进行变换？
switch(coord_select)
   {
   case TRANSFORM_LOCAL_ONLY:
      {
      // 变换局部/模型坐标，并用变换结果替换原来的坐标
      for (int vertex = 0; vertex < 4; vertex++)
       {
       POINT4D presult; // 用于存储变换结果

       // 对顶点进行变换
       Mat_Mul_VECTOR4D_4X4(&root->wall.vlist[vertex].v, mt, &presult);

       // 将结果存回去
       VECTOR4D_COPY(&root->wall.vlist[vertex].v, &presult);

       // 必要时对顶点法线进行变换
       if (root->wall.vlist[vertex].attr & VERTEX4DTV1_ATTR_NORMAL)
         {
         // 对法线进行变换
         Mat_Mul_VECTOR4D_4X4(&root->wall.vlist[vertex].n,
                    mt, &presult);

         // 将结果存回去
         VECTOR4D_COPY(&root->wall.vlist[vertex].n, &presult);
         } // end if
         } // end for index
      } break;

case TRANSFORM_TRANS_ONLY:
   {
   // 对变换后的坐标进行变换，并用变换结果替换原来的坐标
   // 数组 vlist_trans[]用于存储累积变换结果
   for (int vertex = 0; vertex < 4; vertex++)
    {
    POINT4D presult; // 用于存储变换结果

    // 对顶点进行变换
    Mat_Mul_VECTOR4D_4X4(&root->wall.tvlist[vertex].v, mt, &presult);

    // 将结果存回去
    VECTOR4D_COPY(&root->wall.tvlist[vertex].v, &presult);

    // 必要时变换顶点法线
    if (root->wall.tvlist[vertex].attr & VERTEX4DTV1_ATTR_NORMAL)
      {
      // 对法线进行变换
      Mat_Mul_VECTOR4D_4X4(&root->wall.tvlist[vertex].n,
                 mt, &presult);

      // 将结果存回去
      VECTOR4D_COPY(&root->wall.tvlist[vertex].n, &presult);
      } // end if
     } // end for index
```

```
        } break;

      case TRANSFORM_LOCAL_TO_TRANS:
        {
        // 将局部/模型坐标进行变换，并将结果存储到数组 tvlist[]中
        for (int vertex=0; vertex < 4; vertex++)
          {
          POINT4D presult; // 用于存储变换结果

          // 对顶点进行变换
          Mat_Mul_VECTOR4D_4X4(&root->wall.vlist[vertex].v, mt,
                    &root->wall.tvlist[vertex].v);

          // 必要时对法线进行变换
          if (root->wall.tvlist[vertex].attr & VERTEX4DTV1_ATTR_NORMAL)
            {
            // 对法线进行变换
            Mat_Mul_VECTOR4D_4X4(&root->wall.vlist[vertex].n, mt,
                      &root->wall.tvlist[vertex].n);
            } // end if

           } // end for index
        } break;

      default: break;

      } // end switch

  // 变换 front 子树
  Bsp_Transform(root->front, mt, coord_select);

  } // end Bsp_Transform
```

该函数接受三个参数：BSP 树的根、变换矩阵和变换选择标记，它以递归的方式执行指定的变换。然而，在大多数情况下，不需要对 BSP 树进行平移、旋转或变换，因为 BSP 树很可能表示的是静态的室内（或室外）游戏关卡数据。在游戏关卡中，只有相机是移动的。

下面的函数删除 BSP 树，它遍历 BSP 树并释放其占用的内存：

```
void Bsp_Delete(BSPNODEV1_PTR root)
{
// 这个函数递归地删除 BSP 树中的所有节点，并将其占用的内存归还给操作系统

BSPNODEV1_PTR temp_wall; // 临时墙面

// 检测是否为叶节点
if (root==NULL)
  return;

// 删除 back 子树
Bsp_Delete(root->back);

// 删除当前节点，但在此之前先保存 front 子树
temp_wall = root->front;

// 释放内存
free(root);

// 将 root 指向为 front 子树
root = temp_wall;

// 删除 front 子树
Bsp_Delete(root);

} // end Bsp_Delete
```

要调用该函数，只需传递 BSP 树的根即可，这样将递归地删除整个 BSP 树。

最后处于诊断的目的，下列函数用于打印整个 BSP 树：

```
void Bsp_Print(BSPNODEV1_PTR root)
{
// 这个函数以顺序递归的方式遍历 BSP 树，并将每个节点的信息打印到屏幕上

// 检测是否为叶节点
if (root==NULL)
  {
  Write_Error("\nReached NULL node returning...");
  return;
  } // end if

// 遍历 back 子树
Write_Error("\nTraversing back sub-tree...");

// 递归调用
Bsp_Print(root->back);

// 访问当前节点
Write_Error("\n\n\nWall ID #%d",root->id);

Write_Error("\nstate   = %d", root->wall.state);      // 状态
Write_Error("\nattr    = %d", root->wall.attr);       // 属性
Write_Error("\ncolor   = %d", root->wall.color);      // 颜色
Write_Error("\ntexture = %x", root->wall.texture);  // 纹理指针
                            // 用于简单纹理映射
Write_Error("\nmati    = %d", root->wall.mati); // 材质索引，-1 表示没有材质

Write_Error("\nVertex 0: (%f,%f,%f,%f)",root->wall.vlist[0].x,
                    root->wall.vlist[0].y,
                    root->wall.vlist[0].z,
                    root->wall.vlist[0].w);

Write_Error("\nVertex 1: (%f,%f,%f, %f)",root->wall.vlist[1].x,
                    root->wall.vlist[1].y,
                    root->wall.vlist[1].z,
                    root->wall.vlist[1].w);

Write_Error("\nVertex 2: (%f,%f,%f, %f)",root->wall.vlist[2].x,
                    root->wall.vlist[2].y,
                    root->wall.vlist[2].z,
                    root->wall.vlist[2].w);

Write_Error("\nVertex 3: (%f,%f,%f, %f)",root->wall.vlist[3].x,
                    root->wall.vlist[3].y,
                    root->wall.vlist[3].z,
                    root->wall.vlist[3].w);

Write_Error("\nNormal (%f,%f,%f, %f), length=%f",root->wall.normal.x,
                        root->wall.normal.y,
                        root->wall.normal.z,
                        root->wall.nlength);

Write_Error("\nTextCoords (%f,%f)",root->wall.vlist[1].u0,
                  root->wall.vlist[1].v0);

Write_Error("\nEnd wall data\n");

// 遍历 front 子树
Write_Error("\nTraversing front sub-tree..");

Bsp_Print(root->front);

} // end Bsp_Print
```

要调用该函数，只需传递 BSP 树的根即可，但在此之前务必调用 Open_Error_File() 打开一个错误通道。至此，我们介绍了 BSP 系统的方方面面，接下来讨论如何将其加入到图形流水线中。

13.3.9 将 BSP 树集成到图形流水线中

将 BSP 树技术加入到图形流水线中并不难，我们将像集成 z 缓存系统那样来加入 BSP 系统。不需要处理物体的函数，因为 BSP 树中包含游戏世界中所有静态组成部分的几何信息。处理 BSP 树的方式与处理其他物体相同，但需要注意的一点是：必须首先将 BSP 树插入到渲染列表中，且不能对渲染列表进行 z 排序，因为这样做将破坏 BSP 树的排列顺序。

当然，如果读者并不想利用 BSP 树具有的从后到前顺序访问特性，而是要将其用于碰撞检测或剔除（稍后讨论），可以以任何顺序（物体之前或之后）插入 BSP 树中的多边形，并启用 z 排序和/或 z 缓存等。事实上，在基于硬件的游戏中，99% 都不使用 BSP 树来确定渲染顺序，而是使用它们来进行碰撞检测和剔除。然而，在我们的演示程序中，将按从后到前的顺序遍历 BSP 树，以证明使用 BSP 树可以按从后到前的顺序渲染多边形。

要集成 BSP 功能，只需在渲染循环的开头调用函数 Bsp_Insertion_Traversal_RENDERLIST4DV2()，而不是调用将物体插入到渲染列表中的函数。然而，BSP 树中的多边形将按从后到前的顺序被插入，同时加入到渲染列表中的其他东西将位于这些多边形的前面，因为不存在排序的问题。当然，可以启用 z 缓存，但这有悖于这个例子中使用 BSP 树的初衷（确定从后到前的渲染顺序）。下面的例子说明了在渲染流水线中使用 BSP 功能时的操作顺序（摘自稍后将介绍的演示程序，并删除了无关的代码，让读者能够看到主要的函数调用）：

```
// 启动定时时钟
Start_Clock();

// 清空绘制面(drawing surface)
DDraw_Fill_Surface(lpddsback, 0);

// 读者键盘和其他设备输入
DInput_Read_Keyboard();

// 游戏逻辑...

// 重置渲染列表
Reset_RENDERLIST4DV2(&rend_list);

// 创建相机矩阵
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
// 将 BSP 树中的多边形插入到渲染列表中
Bsp_Insertion_Traversal_RENDERLIST4DV2(&rend_list,
                    bsp_root, &cam, 1);

// 执行世界坐标到相机坐标变换
World_To_Camera_RENDERLIST4DV2(&rend_list, &cam);

// 裁剪多边形
Clip_Polys_RENDERLIST4DV2(&rend_list,
   &cam,
   ((x_clip_mode == 1) ? CLIP_POLY_X_PLANE : 0) |
   ((y_clip_mode == 1) ? CLIP_POLY_Y_PLANE : 0) |
   ((z_clip_mode == 1) ? CLIP_POLY_Z_PLANE : 0) );

// 对整个场景进行光照处理
if (lighting_mode==1)
  {
  Transform_LIGHTSV2(lights2, 4, &cam.mcam, TRANSFORM_LOCAL_TO_TRANS);
  Light_RENDERLIST4DV2_World2_16(&rend_list, &cam, lights2, 4);
  } // end if

// 执行相机坐标到透视坐标变换
```

```
Camera_To_Perspective_RENDERLIST4DV2(&rend_list, &cam);

// 执行屏幕变换
Perspective_To_Screen_RENDERLIST4DV2(&rend_list, &cam);

// 锁定后缓存
DDraw_Lock_Back_Surface();

// 设置渲染场境，不使用 z 缓存
rc.attr = RENDER_ATTR_NOBUFFER | RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE;

rc.video_buffer  = back_buffer;
rc.lpitch        = back_lpitch;
rc.mip_dist      = 0;
rc.zbuffer       = (UCHAR *)zbuffer.zbuffer;
rc.zpitch        = WINDOW_WIDTH*4;
rc.rend_list     = &rend_list;
rc.texture_dist  = 0;
rc.alpha_override = -1;

// 对场景进行渲染
Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16(&rc);
// 解除对后缓存的锁定
DDraw_Unlock_Back_Surface();

//  交换前后缓存
DDraw_Flip2();
```

正如读者看到的，除了调用一个函数来插入 BSP 树外，调用的其他 3D 流水线函数与以往相同。BSP 演示程序中包含天花板网格和地面网格（实际上是 OBJECT4DV2），首先将它们插入到渲染列表中，然后再插入 BSP 树，因为根据定义，地面和天花板可能被墙面所遮掩。另外，只要天花板和地面各自是共面的，便无需对其进行排序，稍后将讨论这一点。

13.3.10　BSP 关卡编辑器

在 BSP 演示程序中，大约 90%为界面代码，同时包含几个对 BSP 函数的调用。它淋漓尽致地说明了编写界面有多复杂：在大多数情况，界面代码是其封装的程序代码的 10 倍。该演示程序名为 DEMOII13_1.CPP|EXE，它让用户能够使用鼠标来绘制由 2D 墙面构成的游戏关卡的俯视图，图 13.25 是其运行时的屏幕截图。正如读者看到的，该程序的界面主要由三部分组成：右边为控制区域，左边为编辑区域，最顶上的菜单栏用于选择模式和编译、保存以及加载文件等。

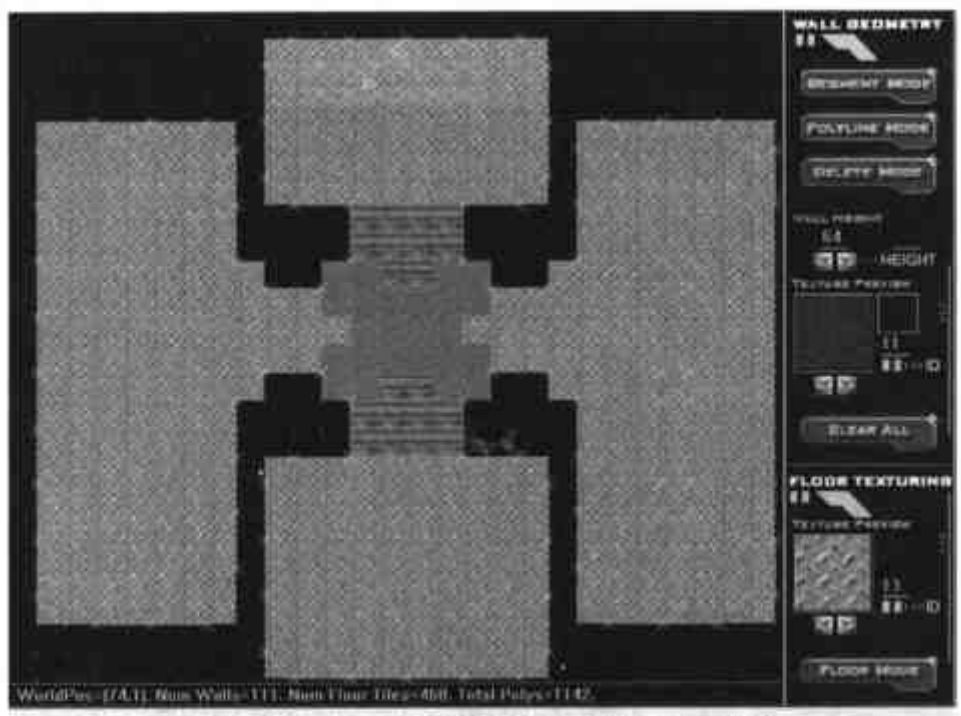


图 13.25　BSP 演示程序运行时的屏幕截图

注意：要编译该演示程序，需要 DEMOII13_1.CPP、所有的库模块（T3DLIB1-11.CPP|H）以及 DirectX .LIB

文件。另外，还需要用作菜单等的资源文件，该文件名为 DEMOII13_1.RC。

讨论如何使用该编辑器之前，先介绍使用它时必须遵循的几个约束条件：

- 线段数不能超过 256 条。
- 绘制墙面时，长度不要超过 2～4 个栅格，否则纹理映射将发生扭曲（由于为提高速度使用了仿射纹理映射）。作者发现，在标准第一人称射击游戏中，墙面长度通常为 2～3 个栅格，高度为 128 个栅格。
- 尽可能避免墙段相互贯穿。
- 让关卡尽可能简单——该演示程序一点也不严密，很容易崩溃。
- 编辑器和查看器都在 Windows 模式下运行，在台式机上运行时，要求的最低分辨率为 800×600。当然，和本书中其他的 Windows 应用程序一样，色深必须为 16 位。

1. Doom 演示程序

该演示程序一点也不像 Doom，但这个名称容易记住，因此使用了它。介绍该编辑器的代码和功能之前，先介绍一下如何加载游戏关卡以及在其中移动，步骤如下：

（1）运行可执行文件 DEMOII13_1.EXE。为确保界面完美无缺，务必将颜色模式设置为 16 位，分辨率设置为 800×600 或更高——建议设置为 1024×768。结果如图 13.26 所示。



图 13.26　运行关卡编辑器

（2）选择菜单 File/Load .LEV File，如图 13.27 所示。然后在文本框中输入 DOOM01.LEV，如图 13.28 所示，然后单击 OK 按钮。



图 13.27　加载关卡

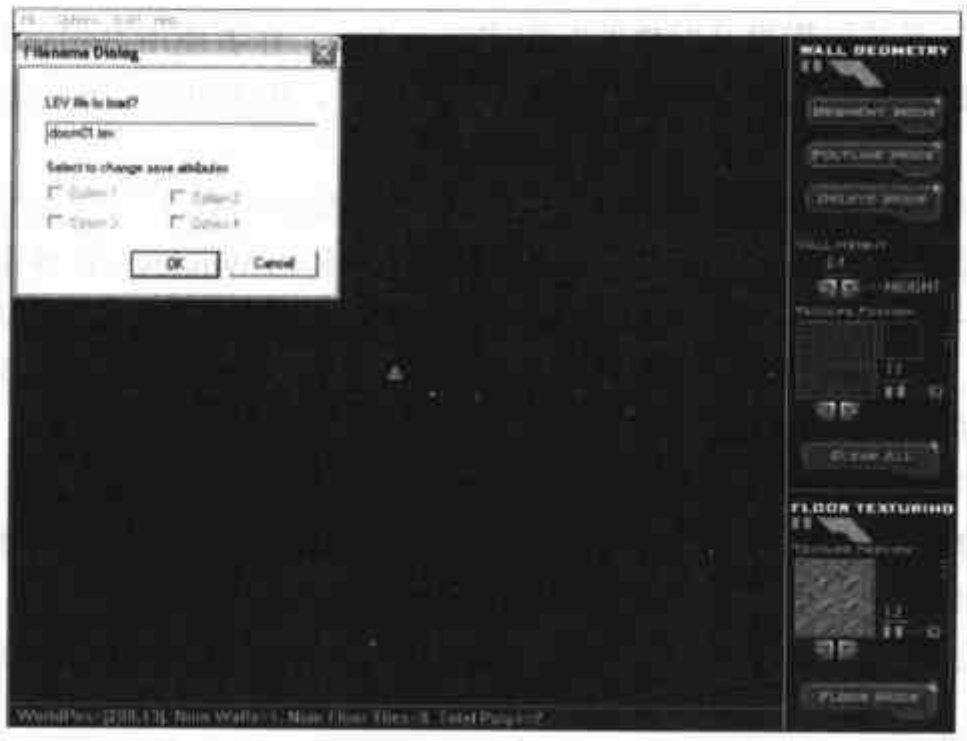


图 13.28　输入关卡文件名

（3）关卡编辑器将加载这个关卡，结果如图 13.25 所示。要将该关卡编译成 BSP 树并查看它，只需使用鼠标选择菜单 Build/Compile .BSP and View，系统将把该关卡编译成 BSP 树，然后切换到 3D 模式，结果如图 13.29 所示。

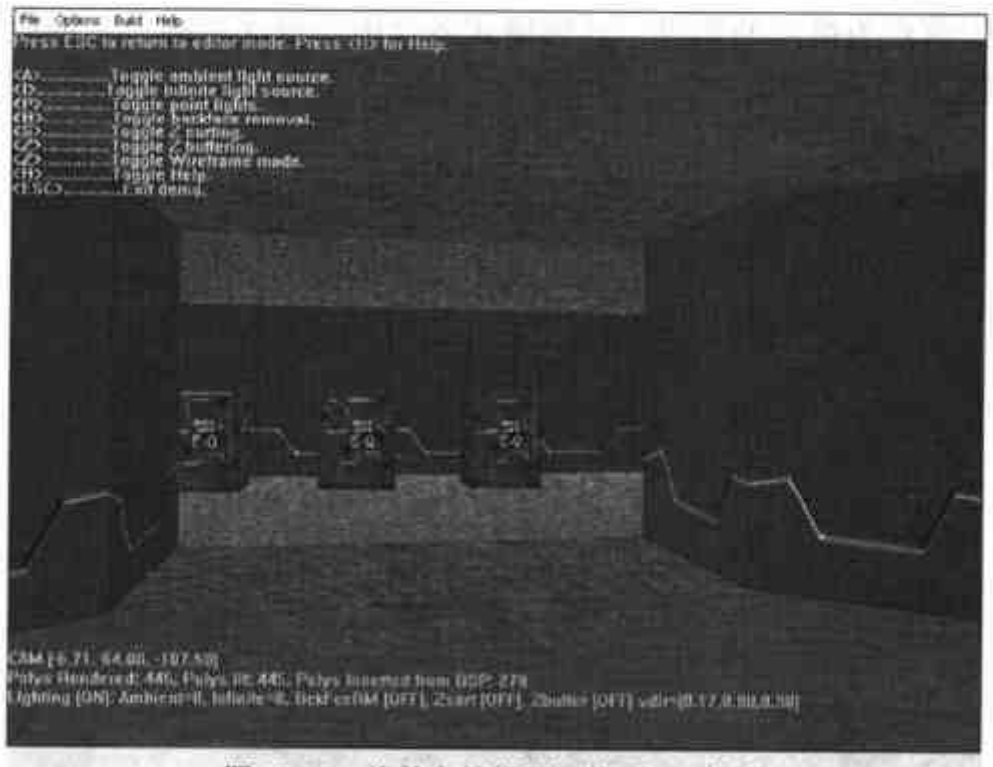


图 13.29　将关卡编辑器切换到 3D 模式

（4）尝试在关卡中移动，按 A、I 和 P 键开/关光源。如果读者的计算机速度足够快（1.5GHz 或更高），显示将瞬时完成。请尝试在关卡中移动，按 W 键启用/禁用线框模式，并查看每帧的多边形数量。

（5）按 Esc 键返回编辑器界面。重复第 3～5 步，不断进入并退出 3D 模式，这是建模的步骤。

警告：该 BSP 演示程序有几个未解决的 bug：有时候，有些墙面会导致系统挂起或不能正确地显示。因此，请确保关卡尽可能简单，否则应用程序可能终止。

读者对该关卡编辑器的功能有了大概了解后，接下来介绍文件格式、各个控件以及如何使用该编辑器。

2. 关卡编辑器的 GUI

决定编写一个基本的关卡编辑器后，作者一开始想使用 Quake 文件格式，但它太复杂了。这种格式涉及的东西太多，而作者喜欢使用简单的文件格式，这有助于读者理解底层的概念，而不是将注意力集中在复杂的文件格式上。

作者咬紧牙关决定编写一个关卡编辑器后，必须决定如何编写它：使用 Visual Basic、Visual C++、Borland

Builder、DirectX、GDI 还是其他程序开发软件？在实际工作中，作者使用 Borland Builder 或 Visual Basic 来编写所有的工具，但就该关卡编辑器而言，这样做太麻烦了，也不方便读者对新增功能的理解，因此使用了 Visual C++。

接下来的问题是，应将其编写成 Windows 应用程序还是全屏应用程序。为支持 3D 模式，要求为 DirectX 应用程序；但为编写 GUI，需要使用按钮和控件。作者决定，使用 Windows 控件来提供菜单，其他界面元素则通过手工来完成——即创建自己的按钮，将其用作编辑区域中的控件。与让 Windows 和 DirectX 和平共处相比，这容易得多。总之，上述问题得到了妥善的解决，该编辑器足以满足我们的要求。有关代码稍后再回过头来介绍，现在介绍用户模式下各控件的用途。

该编辑器的目标很简单：让用户能够在 x-z 平面中绘制墙段，它们表示 3D 世界中的 3D 墙面。另外，用户还能够绘制带纹理的地面砖，在 3D 世界中，地面砖使用多边形表示的。因此，从用户的角度看，只需绘制带纹理的墙段和地面砖，然后构建并查看 BSP 树——这非常简单。

下面介绍界面中右边的控件面板上各个控件。如图 13.30 所示，有两个主要的控件组子面板：

- Wall Geometry；
- Floor Texturing。

（1）线段模式

首先介绍 Wall Geometry 面板。图 13.30 中的元素 1（SEGMENT MODE）用于指定要在编辑区域中绘制单条线段。用户单击该按钮后，系统将切换到单线段模式。这种模式的运行方式如下：用户在编辑区域中移动并单击鼠标后，将创建一个节点；松开鼠标按钮后，用户将看到一条随鼠标移动的线段；再次单击鼠标时，将生成一个以当前点为终点的墙段。这种模式适合用于每次绘制一个墙段。如果中途想放弃当前墙段，只需单击鼠标右键。

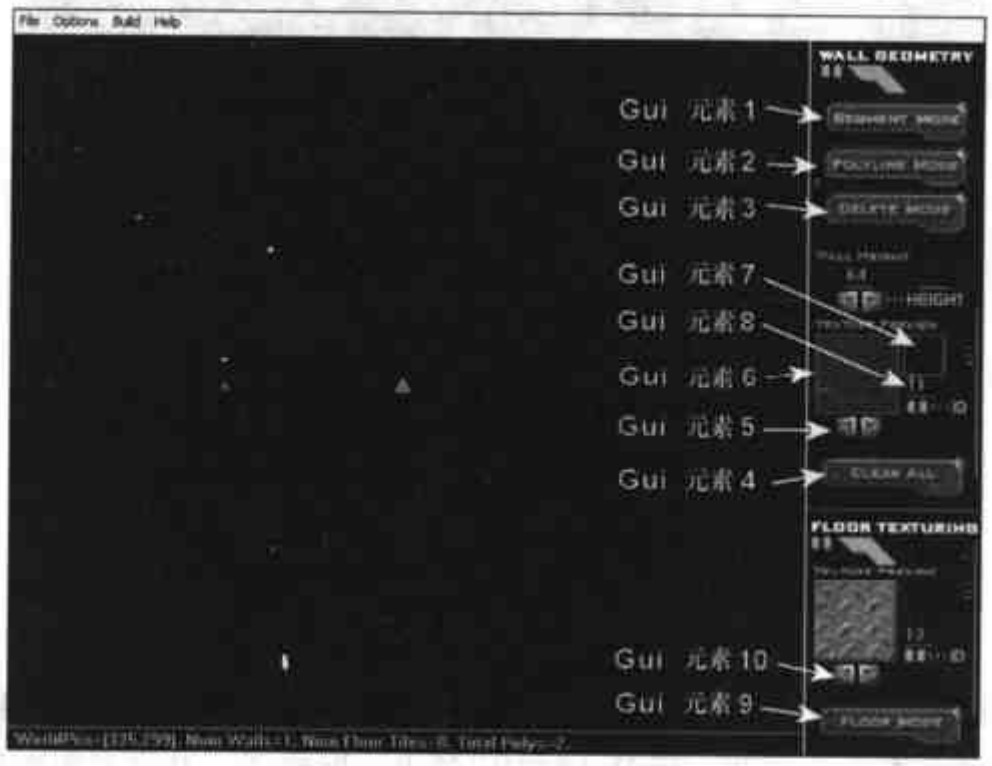


图 13.30　关卡编辑器的界面

（2）折线模式

折线更灵活，在这种模式下，可以连续绘制多条线段。在图 13.30 中，POLYLINE MODE 按钮被标记为元素 2，用户单击该按钮后，编辑器将切换到折线模式。这种模式的运行方式如下：用户在编辑区域中移动并单击鼠标后，将创建一个“起点”；用户松开鼠标按钮并移动鼠标时，将显示下一条线段，要绘制它，只需单击鼠标。用户可以不断地添加线段，要终止折线绘制，只需单击鼠标右键。

（3）删除模式

接下来是墙面删除模式。要切换到这种模式，可单击按钮 DELETE MODE——图 13.30 中的元素 3。进入这种模式后，系统将跟踪编辑区域中离鼠标最近的线段，并通过闪烁来指出这一点。用户只需单击鼠标，便可以删除这条线段。

（4）删除全部

图 13.30 中的元素 4 为 CLEAR ALL 按钮，用于删除整个关卡（包括墙面和地面砖），并重置到初始状态。要删除整个关卡，只需单击该按钮即可。

（5）墙面高度和纹理预览

至此读者知道了如何在编辑器中绘制线段，但它们表示的是什么呢？它们表示垂直于 x-z 面（世界空间）的 3D 墙面，其中每个栅格大约相当于 48 个单位。然而，还没有介绍如何指定墙面的高度和纹理。这些信息是通过 GUI 元素 5 和 6 来控制的。GUI 元素 5 用于控制墙面高度，用户可以单击数值下面的蓝色箭头来修改墙面高度。作者建议将墙面的高度和长度分别设置为 128 和 2 个栅格左右。

对于每个墙面，都有一个纹理被映射到其表面，如果纹理太小，则将其拉伸，而不重复。纹理的大小皆为 128×128，有大约 20 个纹理可供选择。当前使用的纹理显示在 GUI 元素 6 指向的纹理预览窗口中。要更换纹理，可单击预览窗口下面的按钮。所有墙体都使用当前选择的高度和纹理，而不能修改。要修改墙面，惟一的办法是删除它再重新创建。

（6）墙面高度和纹理扫描器

除墙面高度和纹理预览控件外，还有墙面高度和纹理扫描器——图 13.30 中的元素 7 和 8。它们类似于绘图程序中的吸管（eye dropper），用于显示编辑区域中离鼠标最近的墙面的高度和纹理。例如，假设读者绘制一个关卡后，忘记了某个墙面的纹理或高度，只需将鼠标在该墙面上移动，直到墙面闪烁，墙面高度和纹理扫描器将显示这些信息。别忘了，您不能修改墙面，而只能将其删除再重新创建。

（7）地面砖模式

在该关卡编辑器的 GUI 中，FLOOR MODE 按钮是除菜单栏外的最后一个重要组成部分，在图 13.30 中，这被标记为元素 9。绘制关卡时，默认情况下没有地面和天花板。要加入地面砖，必须单击 FLOOR MODE 按钮，以切换到地面砖模式；然后在要加入地面砖的地方单击鼠标，显示在 Floor Texturing 面板的纹理预览窗口中的纹理将被用于该地面砖。如果加入的地面砖不对，可在其上单击鼠标右键来将其删除，但这必须在地面砖模式下才管用。和墙面一样，可以从大约 20 种纹理中选择用于地面砖的纹理，为此可单击地面砖纹理预览窗口下面的左箭头按钮或右箭头按钮（GUI 元素 10）。

（8）菜单栏

菜单栏中包含多个菜单，下面依次介绍它们：

- File——包含三个菜单项：Load .LEV File、Save .LEV File 和 Exit。前两个菜单项用于分别加载和保存 ASCII 格式的 BSP 文件，用户只需输入要加载或保存的文件名（扩展名为.LEV），然后单击 OK 按钮即可；菜单项 Exit 退出程序。
- Options——用于指定要在编辑区域显示哪些东西。有时候，用户可能只想显示墙面或地面砖，或者不想实现栅格，Options 菜单提供了这些功能。该菜单包含三个菜单项：View Grid、View Walls 和 View Floor。通过选择这些菜单项，可以在显示和不显示之间切换。
- Build——该菜单包含两个菜单项，但只有一个是可用的——Compile BSP and View。它用于编译 BSP 关卡并切换到 3D 渲染模式。在 3D 渲染模式下，用户可以按箭头键在关卡中移动，还可以按 H 键来显示帮助菜单。要返回到编辑模式，可按 ESC 键。
- Help——该菜单只包含一个菜单项——About。

注意：相机将被放置在 3D 世界的(0，0、0)处，用一个绿色三角形表示。另外，观察者的高度为当前墙面高度，因此切换到 3D 模式之前，务必将当前墙面高度设置为小于最大的墙面高度，否则观察者的头部将穿过天花板。

（9）有关编辑器和查看关卡的最后说明

有关该编辑器的用法就介绍到这里，但讨论技术方面的内容之前，有必要对几个细节进行说明。在大多数情况下，用户将启动该编辑器，其中包含一个空的关卡；如果要编辑关卡，可使用菜单 File/Load .LEV File 加载它。无论在哪种情况下，用户都将选择墙面高度和纹理，然后将墙面加入到关卡中。然后，用户可能切换到地面砖模式，并加入一些地面砖。要查看关卡，用户修改墙面高度，以指定玩家的视点高度，

然后选择菜单 Build/Compile BSP and View，以生成 BSP 并在关卡中漫游。最后，按 Esc 键，以退出 3D 模式，返回到编辑模式。

用户将不断重复上述过程，直到关卡满意为止。有几点读者必须牢记。首先，编辑器将根据用户铺设的地面砖来生成地面和天花板，天花板为地面的倒影，被渲染在关卡中最大墙面高度对应的位置处。另外，建议读者在编译关卡之前将其存盘，因为处理复杂的关卡时，编辑器有时会挂起。

3. BSP 编辑器使用的文件格式

读者知道编辑器的用法后，下面介绍关卡数据是如何存储的。使用的文件格式的扩展名为.LEV，它是一种可供人类阅读的 ASCII 格式，由文件头（header）、墙面列表和地面砖组成。下面是图 13.31 所示的方形房间的文件，该文件名为 BSPEXAMPLE01.LEV，读者可在附带光盘中找到它。

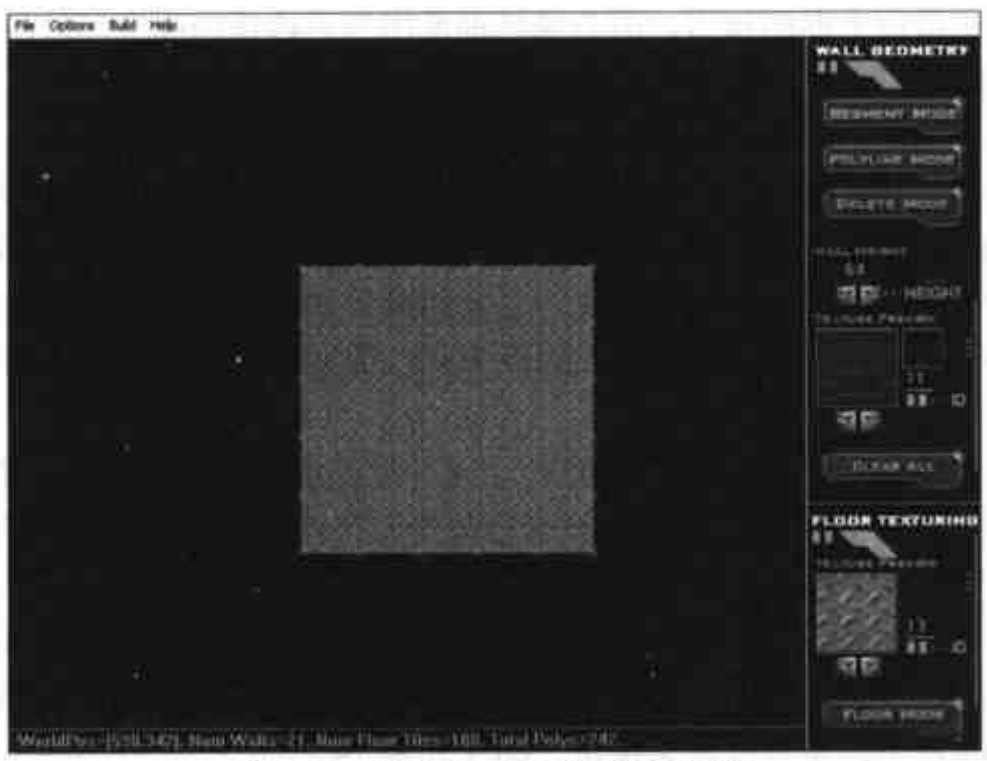


图 13.31　只包含一个方形房间的关卡

```
Version: 1.0

NumSections: 2

Section: walls

NumWalls: 20

240 192 288 192 0 128 11 65535 4169
288 192 336 192 0 128 11 65535 4169
336 192 384 192 0 128 11 65535 4169
384 192 432 192 0 128 11 65535 4169
432 192 480 192 0 128 11 65535 4169
480 192 480 240 0 128 11 65535 4169
480 240 480 288 0 128 11 65535 4169
480 288 480 336 0 128 11 65535 4169
480 336 480 384 0 128 11 65535 4169
480 384 480 432 0 128 11 65535 4169
480 432 432 432 0 128 11 65535 4169
432 432 384 432 0 128 11 65535 4169
384 432 336 432 0 128 11 65535 4169
336 432 288 432 0 128 11 65535 4169
288 432 240 432 0 128 11 65535 4169
240 432 240 384 0 128 11 65535 4169
240 384 240 336 0 128 11 65535 4169
240 336 240 288 0 128 11 65535 4169
240 288 240 240 0 128 11 65535 4169
240 240 240 192 0 128 11 65535 4169

EndSection
```

```
Section: floors

NumFloorsX: 27
NumFloorsY: 24

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 13 13 13 13 13 13 13 13 13 13 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

EndSection
```

文件的开头为文件的版本号，这里为 1.0。接下来为区段（section）数。当前，所有文件的区段数都为 2（墙面部分和地面部分），但我们将使用这种文件格式，区段数字段提供了扩展功能。作者不喜欢通过对文件进行分析来确定需要分析多少数据，而希望被告知。创建自己的文件格式时，这是一种不错的特性——告诉加载函数，后面有哪些数据。接下来为区段，它们可以包含任何数据，但格式总是这样的：

```
Section: "section name"
.
.
.
EndSection
```

其中的点表示数据，可以是任何东西。区段名告诉加载程序，其中包含的什么信息。在这个例子中，有两种区段：walls 和 floors。墙面区段在前，其中包含一个这样的字段：

```
NumWalls: 20
```

这指出了需要逐行地读取多少个墙面。当前，每个墙面包含 7 个值，格式如下：

```
x0.f y0.f x1.f y1.f elev.f height.f text_id.d color.d attr.d
```

其中 x0.f、y0.f、x1.f 和 y1.f 是线段端点的坐标；evel.f 和 hight.f 是墙面的仰角（elevation）和高度（当前没有使用仰角，其值总是为 0）；最后，text_id.d、color.d 和 attr.d 分别是纹理 ID、颜色和属性。

注意：.f 表示浮点数，.d 表示整数。

例如，在 BSPEXAMPLE01.LEV 中，第一个墙面的数据如下：

```
240 192 288 192 0 128 11 65535 4169
```

这意味着该墙段的起点和终点坐标分别是(240，192)和(288，192)，仰角为 0.0，高度为 128.0，RGB 颜色为 255.255.255，属性为 4169——使用恒定着色处理、双面的、颜色模式为 16 位。

分析完墙面区段后，将遇到关键字 EndSection，然后是地面区段：

```
Section: floors
```

前两行指出了游戏世界的大小，单位为地面砖数（tile）：

```
NumFloorsX: 27
NumFloorsY: 24
```

地面砖是以单个整数存储的，它表示地面砖的纹理 ID；如果为-1，则表示相应的地方没有地面砖。引擎对地面砖进行分析，根据它们在阵列中的位置，生成一个由多边形组成的 OBJECT4DV2，每块地面砖都被划分成两个带纹理的三角形。

有关文件格式就介绍到这里。这种文件格式的优点是，可以使用 ASCII 编辑器对其进行修改，同时可以通过关键字 Section 和 EndSection 添加其他功能，因此非常灵活。

4. 关卡编辑器程序

该关卡编辑器是一个典型的七拼八凑式程序，根本没有遵循清晰编程的理念。尽管如此，它还是管用的，我们以后将以它为基础加入更多的功能，供与 BSP 相关的演示程序使用（如果还会编写这样的演示程序的话）。该关卡编辑器有一组编辑关卡时需要执行的函数，它们看似容易，但编写起来极其费力。下面简要地讨论它们。

首先是 GUI。作者必须编写支持鼠标单击、拖动和绘图的按钮处理程序。这里的问题是，由于混合使用了 DirectX 和 GDI/Win32 菜单，因此必须让 Windows 来处理鼠标事件。这意味着鼠标事件将被传递给 WinProc()而不是 DirectInput()，因此必须将其保存，然后在 Game_Main()循环中读取。另外，由于使用 DirectX 来渲染图像，因此 DirectX 和我们的系统每秒渲染 30 次。换句话说，窗口中的绘图操作是实时进行的，同时重画也是实时的——不是静态图像。这是一个问题，设计编辑器时，必须决定是否要实时地更新图像。

接下来的问题是墙面和地面的绘制。作者使用线段和位图来表示墙面和地面纹理，但是使用鼠标来指定准确的位置比较困难。另外，2D 屏幕表示的是 3D 左手坐标系的俯视图（x-z 平面），因此需要选择缩放比例和范围，用于将 2D 数据转换为 3D 数据时对其进行缩放。这种选择和栅格大小等一样，也是半随意的（semi-arbitrary）。

最后，还需要创建一个用于表示墙面和地面的数据结构，供编辑、加载和保存关卡时使用；而不使用更为复杂的 BSPNODEV1。该数据结构如下：

```
// 下面的数据结构用于表示 2D 线段
typedef struct BSP2D_LINE_TYP
    {
    int id;         // ID
    int color;      // 颜色
    int attr;       // 属性（着色模式等）
    int texture_id; // 纹理索引
    POINT2D p0, p1; // 线段的端点

    int elev, height; // 墙面高度

    } BSP2D_LINE, *BSP2D_LINE_PTR;
```

该结构只供编辑器用来表示 2D 世界，与 3D 引擎毫无关系。对于地面，只需使用一个 2 维整型数组来表示即可：

```
int floors[BSP_CELLS_Y-1][BSP_CELLS_X-1];
```

用户选择菜单 Build/Complie BSP and View 后，编辑器将执行两项任务。首先，将墙面从 BSP2D_LINE 格式转换为一个 BSPNODEV1 链表。这是由函数 Convert_Lines_To_Walls()完成的，其代码太长，这里无法列出；它基本上是将 2D 墙面列表转换为一组 3D 多边形，并将它们存储到一个 BSPNODEV1 链表中。然后，该链表的根被传递给 Bsp_Build_Tree()，以创建 BSP 树。

现在，未处理的只有地面和天花板了。惟一需要生成的网格是地面，因为天花板是地面的平移版本，可以使用同一个 OBJECT4DV2 来表示它们。生成地面的函数如下：

```
int Generate_Floors_OBJECT4DV2(OBJECT4DV2_PTR obj, // 指向物体的指针
        int rgbcolor,     // 没有纹理时地面的颜色
        VECTOR4D_PTR pos,// 初始位置
        VECTOR4D_PTR rot,// 初始旋转角度
        int poly_attr);  // 着色模式等属性
```

它接受 5 个参数：指向输出物体的指针、没有纹理时地面砖的颜色（通常为白色）、初始位置、初始旋转角度和地面网格中所有多边形的属性。例如，如果将如图 13.32a 所示的地面砖传递给该函数，将生成类似于图 13.32b 所示的网格，其中的多边形的纹理坐标、属性等被设置好。

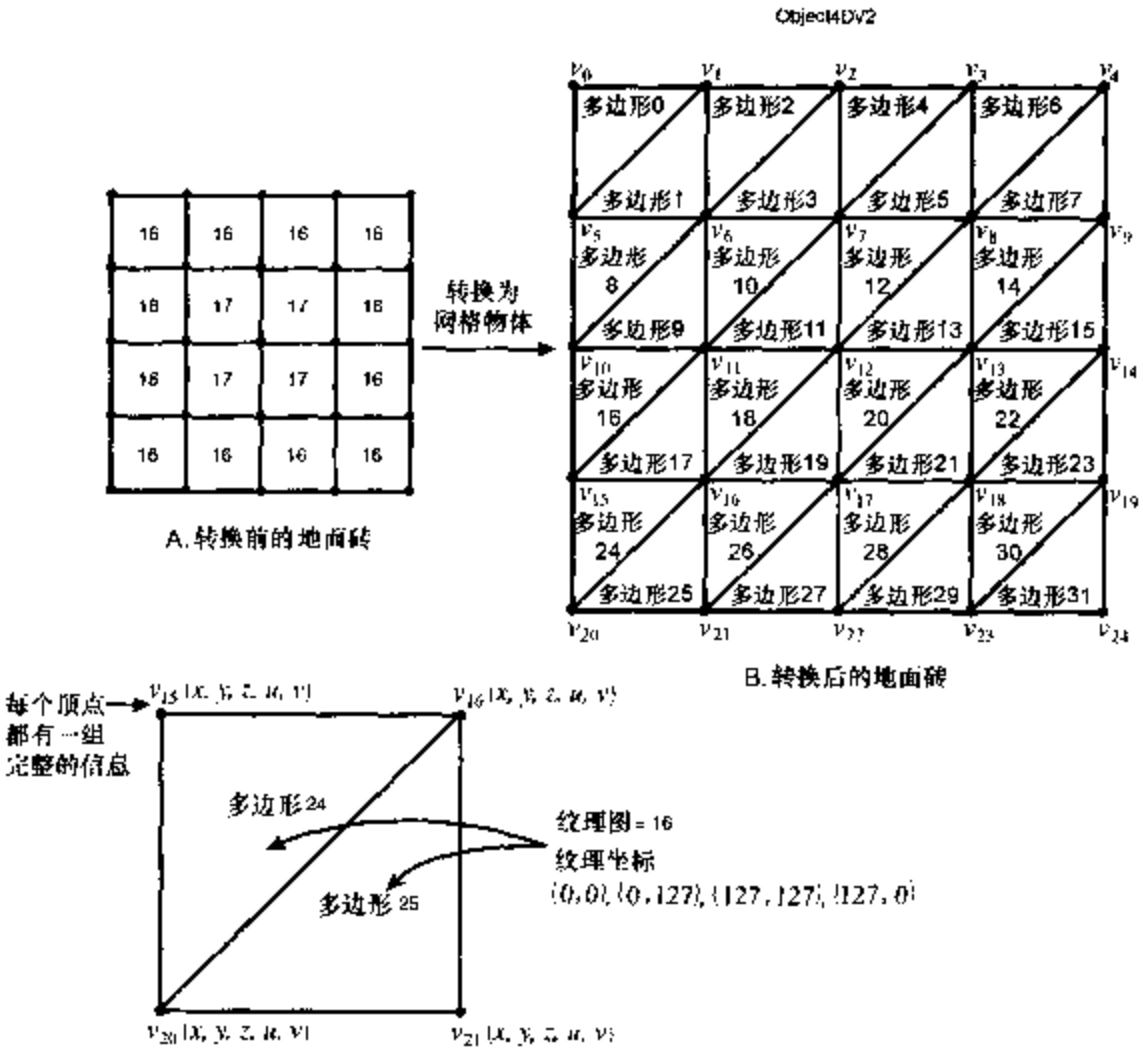


图 13.32　将地面砖转换为网格

生成地面网格后，编辑器切换到 3D 模式，后者类似于我们一直在使用的标准 3D 演示程序模型循环。惟一的差别是，该循环不使用 z 排序或 z 缓存，而是在每帧中按下述顺序将几何体插入到渲染列表中：

1. 地面；
2. 天花板；
3. BSP 树。

只要 BSP 树中多边形树的排列顺序是正确的，这便能保证多边形的渲染顺序是正确的，因为地面和天花板总是位于 BSP 树中多边形的“后面”。

13.3.11　BSP 的局限性

虽然在运行期间，BSP 树确定多边形渲染顺序的速度快得难以置信，但它并非没有局限性。首先，游戏世界必须是半静态的。这意味着多边形不能旋转，但在有些情况下可以平移。例如，如图 13.33a 所示的视图中包含两面墙，如果将其中一面墙往上移到如图 13.33b

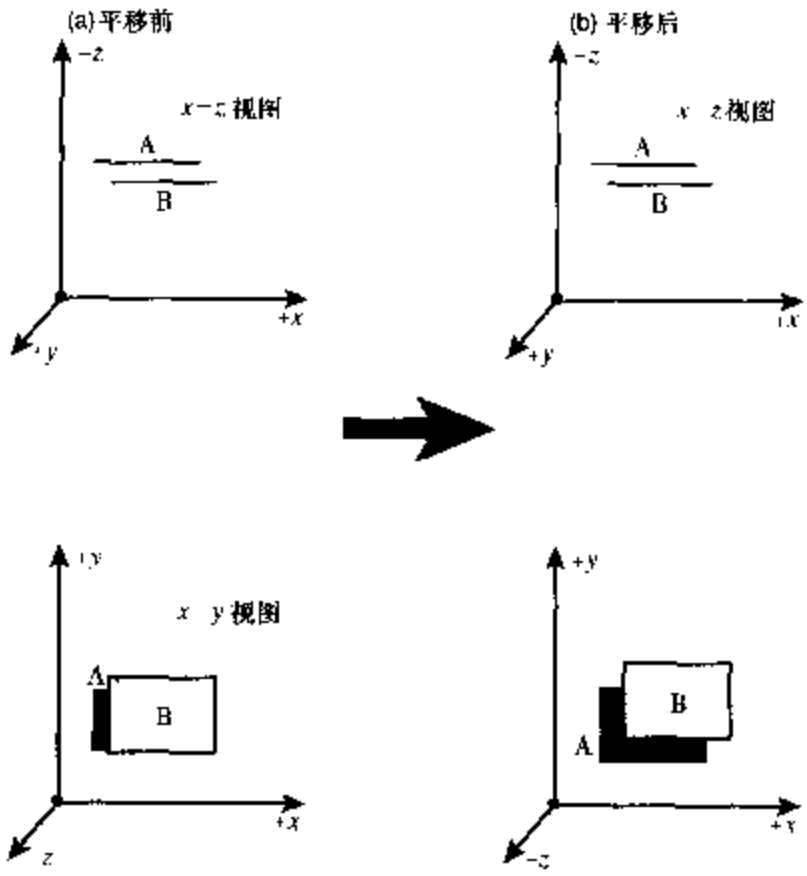


图 13.33　墙面平行于 y 轴平移时 BSP 树保持不变

所示的位置，渲染顺序将保持不变。因此，当多边形在其所在的平面内平移时，仍可以使用 BSP 树，但仅此而已。

BSP 树的第二个重要局限性是，不能用于模拟程序或包含大量移动物体的游戏中，因为在这种情况下，每帧都需要重新生成 BSP 树。然而，对于渲染问题而言，BSP 树是一种很不错的解决方案，可用于玩家在静态环境中移动的情形。BSP 树并非只能用于解决渲染问题。

13.3.12 使用 BSP 树的零重绘策略

给定任何视点，BSP 树可用于确定场景中一组静态多边形从后到前的准确渲染顺序。这是使用一种改进的顺序递归搜索算法，像画家算法那样按从后到前的顺序访问节点实现的。

这种方法存在的问题是，在很多情况下，同一个像素将被绘制多次。这是任何排序或 z 缓存算法都存在的一个大问题：确定正确的渲染顺序后，同一个像素可能被重新绘制很多次（有些情况下，甚至为数百次）。

例如，图 13.34 是一个 x-z 平面视图，其中包含一组多边形。这些多边形已经按从后到前的顺序排列好，或者使用 z 缓存来渲染它们。使用 z 排序时，其中最小的多边形中每个像素都将被其他多边形覆盖；最后次大的多边形中每个像素都将被最大的多边形覆盖。在这个简单的例子中，除最近的多边形外，其他多边形中的每个像素都将被绘制 1 到 n–1 次，其中 n 为多边形的总数。

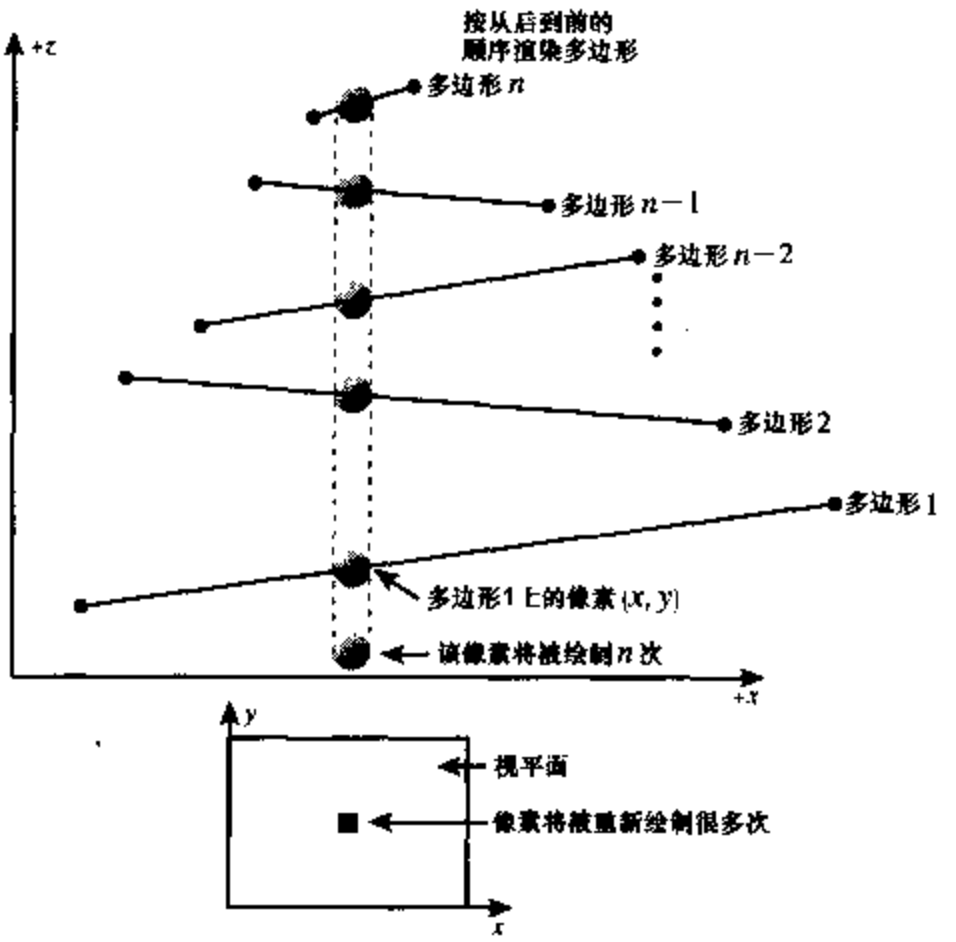


图 13.34 最糟糕的重绘情形

使用 z 缓存也一样糟糕：不但要重绘像素（如果 z 值更小），还必须检测 z 缓存。虽然使用 z 缓存时，重绘不会那么频繁，但每次都需要读取和/或写入 z 缓存。同样，绘制像素或检测看不到的像素的开销让我们承担不起。因此，必须有一种解决重绘问题的方案。

BSP 树是救命稻草：按从前到后而不是从后到前的顺序遍历 BSP 树，对于访问的每个多边形，屏蔽到屏幕的数据写入——将多边形用作模板缓存。从理论上说，这很不错，但实际上这与 z 缓存没有什么区别——检测条件，如果像素满足条件，则将其写入到屏幕中。这样做确实避免了重绘，但仍需要对每个像素进行检测。现在的问题是，能否以更聪明的方式利用从后到前的顺序。答案是肯定的，一种方法是使用扫描线渲染。

扫描线渲染的工作原理如下：不以逐个像素的方式绘制多边形，而是计算每条扫描线的端点，并将其插入到一个链表或数据结构中。基本上，这相当于不使用对每条扫描线进行光栅化的内循环，而是将扫描线端点数据插入到链表中。对于每个多边形，将其各条扫描线插入到链表中，然后根据 x 和 y 对扫描线进

行排序，这样，对于屏幕上的每一行，都有一组扫描线记录。这种处理过程如图 13.35 所示，至此，问题变成了如何处理一组扫描线。如果能够解决单条扫描线的问题，便解决了整个问题，因此我们从这里开始。

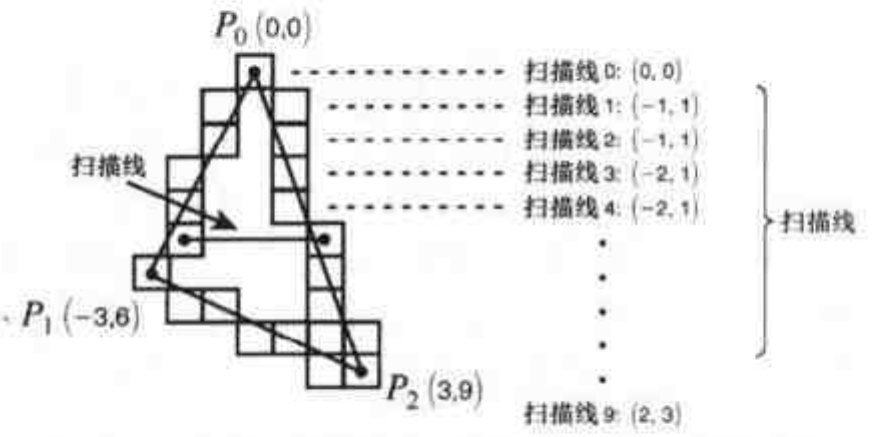


图 13.35　将多边形存储为扫描线而不是动态地渲染它们

下面来看看可能出现的各种情形（如图 13.36 所示）：

- 情形 1：第 n 行只有一条扫描线。在这种情形下，只需光栅化该扫描线即可。

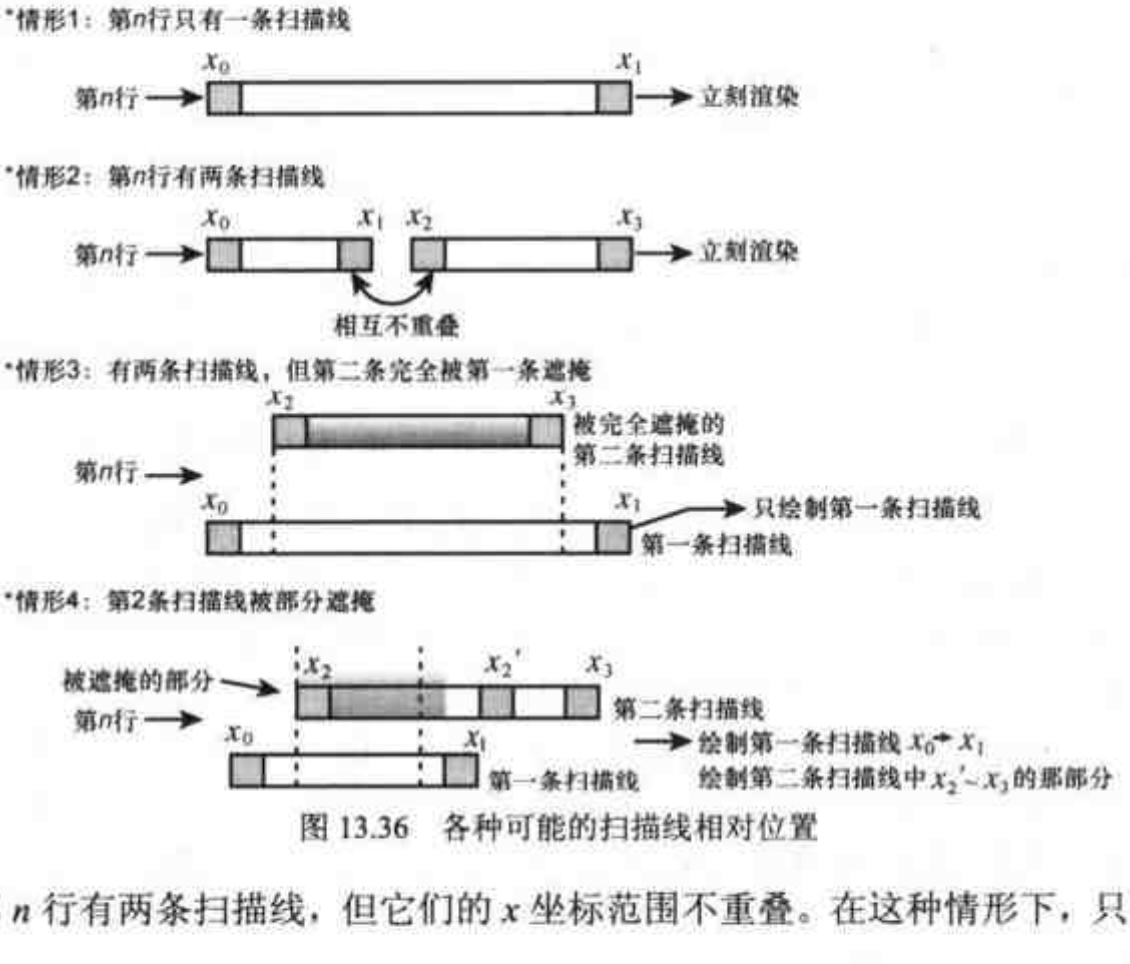


图 13.36　各种可能的扫描线相对位置

- 情形 2：第 n 行有两条扫描线，但它们的 x 坐标范围不重叠。在这种情形下，只需渲染这两条扫描线即可。
- 情形 3：第 n 行有两条扫描线，但第二条完全被第一条遮掩，因此将第二条扫描线丢弃，避免重绘。
- 情形 4：第 n 行有两条扫描线，但第二条被第一条部分遮掩。在这种情形下，根据第一条扫描线对第二条扫描线进行裁剪，然后只绘制第二条扫描线中余下的部分——同样避免了重绘。

结合使用扫描线技术和从前到后渲染技术，可实现完美的渲染，同时避免重绘。然而，这是要付出代价的：判断扫描线是否被遮掩或需要进行裁剪，然后将扫描线合并。在涉及纹理坐标等时，这种算法实现起来并不容易；然而，实现不能完全避免重绘的粗糙版本可能是值得的。

13.3.13　将 BSP 树用于剔除

空间划分算法最强大的功能之一是，能够快速地将大量的多边形信息从流水线中剔除。事实上，在大多数情况下，当前人们只将 BSP 树用于剔除和碰撞检测。凸形子空间特性和遍历的线性时间复杂度使得 BSP 树非常适合用于大规模剔除。这里将介绍两种适用于 BSP 树的剔除技术：背面剔除和视景体剔除，前者是一种原子型（atomic）剔除技术，而后者是一种高级（high-level）剔除技术，它利用 BSP 结构来剔除不可见的多边形。

1．背面剔除

在遍历 BSP 树，将多边形插入到渲染列表中的同时，可以进行背面剔除或对多边形进行裁剪。

然而，在大多数情况下，墙面都是双面的，因此标准的背面剔除操作不管用，因为所有的面都将通过剔除检测。如果要在将 BSP 树插入到渲染列表中的同时删除背面多边形，必须采用一种更精确的背面剔除

算法。

出于复习的目的，图 13.37 说明了标准的背面剔除方法。如图中所示，首先计算（或引用，如果预先计算好了）多边形的面法线，然后计算该法线向量与到视点的向量的点积。如果这两个向量之间的夹角大于 90 度（点积小于零），则表明多边形是背面。然而，正如前面指出的，这种方法仅适用于单面多边形。

另外，这种背面消除算法没有考虑观察方向向量（view direction vector）。例如，在图 13.38 中，法线和视点向量（view vector）之间的夹角小于 90 度，但如果考虑到观察方向向量可以知道，虽然标准的背面消除算法不会将该多边形剔除，但它实际上是不可见的。

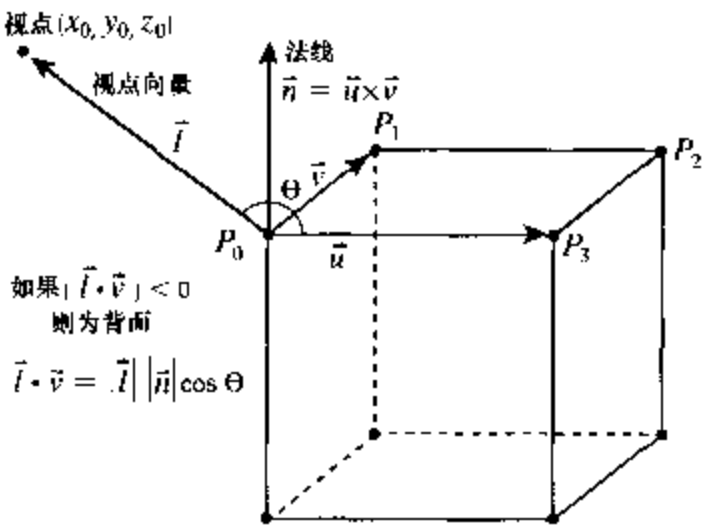


图 13.37　背面消除的详细分析

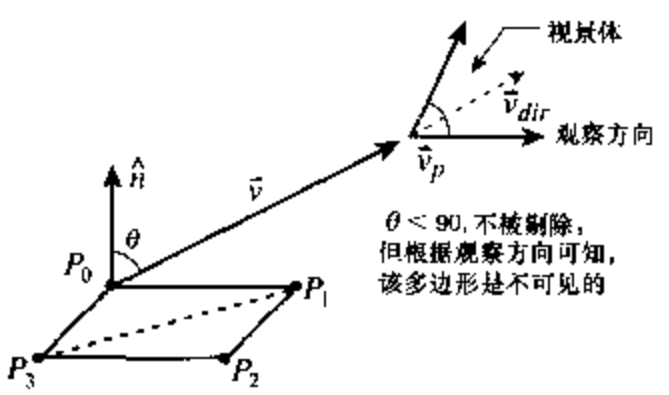


图 13.38　标准背面消除算法没有考虑观察方向

如果在相机空间中执行背面消除，将不会出现这种问题，因为视点向量的方向与+z 轴平行，即为（0，0，1）。在这种情况下，背面剔除将是正确的，不会让多余的多边形传递到流水线的下游。但问题是，仅仅为了剔除就应将多边形变换为相机坐标吗？答案当然是否定的。我们一直是在世界空间中进行背面剔除的，其结果是有些多边形漏网了，在裁剪阶段，这些多边形被简单拒绝。

在大多数情况下，BSP 树中的多边形是双面的，因此在遍历 BSP 树并将其中的多边形插入到渲染列表中时，必须使用更严格的背面剔除算法。当然，可以完全忽略这种优化，将多边形都插入到渲染列表中，像以前一样，让渲染列表背面消除函数来完成这项工作。然而，在下一节剔除整个子空间时，这种精彩的优化将派上用场。

为在世界空间中根据视点位置和观察方向解决双面多边形的背面剔除问题，需要知道哪些信息呢？图 13.39 说明了这种背面剔除方法。

首先，需要判断视点位于多边形的前面还是后面；接下来需要计算多边形面法线与视点向量之间的夹角。然而，还必须考虑视景体的视野，换句话说，即使正面法线与观察方向向量（不要将其同视点向量混为一谈，视点向量是从多边形到视点的向量）大于 90 度，多边形也可能在视野中，这是因为视野可能非常开阔，使得在上述条件下，多边形也可见，如图 13.39 所示。

因此，执行背面消除计算时，需要考虑视野。我们需要计算或使用多边形的面法线，然后计算它与视点向量的点积，以判断视点位于多边形的前面还是后面。

确定视点位于多边形的哪一边后，需要计算多边形这一面的面法线与观察方向向量的点积，并将其作为最终的判断因子。存在两种情形：视点位于多边形的前面时，如果(θ+FOV/2)>90，则多边形是可见的，否则是不可见的；视点位于多边形的后面时，如果(θ-FOV/2)<90，则多边形是可见的，否则不可见。从数学的角度上说，情况如下。

给定：

- 观察方向向量 v_{dir}。
- 视点位置 v_{pos}。
- 多边形面法线向量 n。
- 多边形到视点的向量 v_{view}。
- 面法线 n 和观察方向向量 v_{dir} 之间的交角 θ。

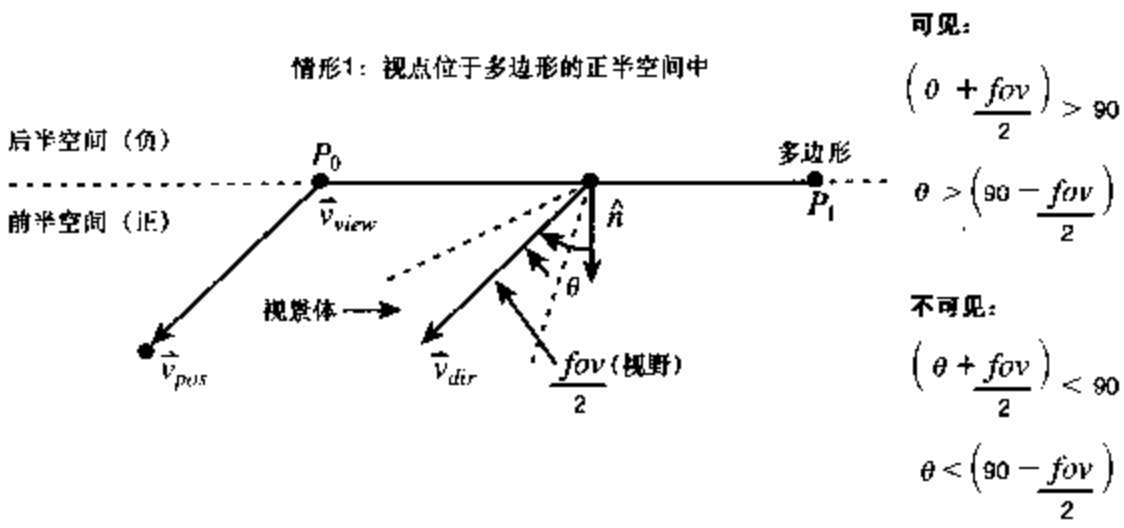


图 13.39　考虑了视点位置和观察方向的完整背面消除算法

情形 1：如果 v_{pos} 位于多边形的前面，即 $n \cdot v_{view}>0$，则：

可见条件为(θ+FOV/2)>90。

$n \cdot v_{dir} = |n| * |v_{dir}| * \cos\theta$

假设这两个向量都被归一化，即$|n| = |v_{dir}| = 1$，则：

$n \cdot v_{dir} = \cos\theta$

对等式两边求反余弦，结果如下：

θ = arccosine ($n \cdot$ vdir)

现在，只需将θ代入可见条件中，便可以确定是否要剔除多边形。然而，使用 arccosine() 显然不能达到实时的效果。稍后将讨论这一点，现在先来看一下视点位于多边形后面的情形。

情形 2：如果视点 v_{pos} 位于多边形的后面，即 $n \cdot v_{view}<0$，则：

可见条件为(θ - FOV/2)<90。

θ 的计算方法与情形 1 中相同。将其代入可见条件，便可以判断多边形是否应剔除。

（1）快速计算反余弦

除非读者使用的是量子计算机，否则计算反余弦的开销是令人承担不起的，因为调用数学库函数 arccosine() 无法达到实时的效果。下面详细地介绍另一种技巧：使用查找表。首先介绍一下 arccosine() 的功能。

arccosine() 计算反余弦，即给定一个位于[–1，1]的值 x，找出函数 $f(x)$，使得 $\cos(f(x)) = x$。$f(x)$ 被称为反余弦。例如：

cosine(30) = .866

arccos(.866) = 30 度

可以创建一个查找表，以便快速地计算反余弦。为此，需要使用反余弦作为映射函数，将[–1，1]的值映射到 0～180 度的角度。我们知道，查找表索引不能为负数和小数，因此首先需要将范围[–1，1]平移 1，变成[0，2]。使用这种索引范围时，查找表只能包含 3 个元素，这几乎毫无用处。我们需要将上述索引范围扩大，使得至少 0～180 的每个整数都有对应的索引。为此，需要将索引范围放大 90 倍，结果为[0，180]。

计算索引的公式如下：

索引 = (x + 1)*90
其中 x 的取值范围为[-1，1]。

接下来我们创建一个查找表，其中包含均匀地分布在[-1，1]内的 181 个值的反余弦，反余弦的取值范围为 0～180 度，如图 13.40 所示。

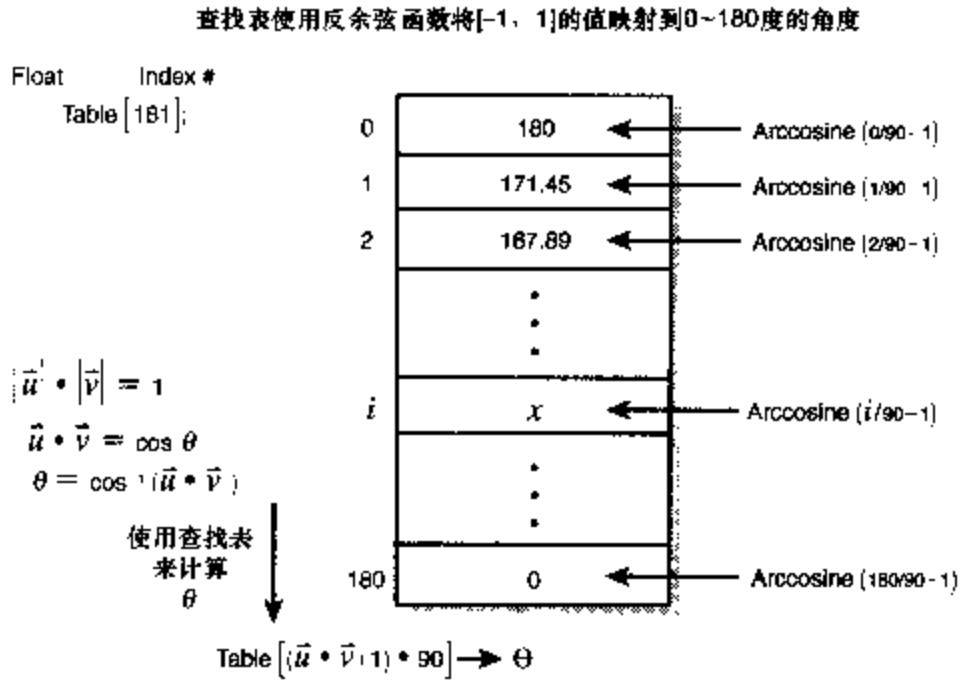


图 13.40　反余弦查找表

要提高精度，可增加查找表中的条目。例如，将[-1，1]分成 360 等分，但反余弦范围仍为 0～180 度。作者编写了一个完成这项工作的函数。首先来看存储查找表的数组：

```
float dp_inverse_cos[360+2];
```

上述代码定义了一个包含 362 个元素的查找表。之所以定义 362 个元素，是为了避免访问第 360 或 361 个元素时，由于映射期间的浮点数溢出导致数组访问越界。为简化工作，作者还定义了下述宏，它用于计算 x 的反余弦，其中 x 的取值范围为[-1，1]：

```
#define FAST_INV_COS(x)\
 (dp_inverse_cos[(int)(((float)x+1)*(float)180)])
```

下面是生成查找表的函数：

```
void Build_Inverse_Cos_Table(float *invcos,   // storage for table
                int  range_scale) // range for table to span

{
// 这个函数创建一个反余弦查找表，用于根据点积计算夹角
// 它将[-1, 1]的值映射到索引范围[0, range_scale]
// 然后将范围[-1, 1]均匀地划分成 rang_scale + 1 份，并计算每个值的反余弦
// 再将结果存储到一个包含 rang_scale + 1 个元素的 float 数组中
// 查找表的精度为 180/rang_scale，例如，如果 rang_scale 为 360，则精度为 0.5 度

float val = -1; // 起始值

// 创建查找表
for (int index = 0; index <= range_scale; index++)
  {
  // 将下一个元素存储到查找表中
  val = (val > 1) ? 1 : val;
  invcos[index] = RAD_TO_DEG(acos(val));

  // 计算下一个[-1, 1]的值
  val += ((float)1/(float)(range_scale/2));

  } // end for index
```

```
// 增加一个元素，以避免访问数组时越界
invcos[index] = invcos[index-1];

} // end Build_Inverse_Cos_Table
```

在代码的初始化部分，这样调用上述函数：

```
// 创建反余弦查找表
Build_Inverse_Cos_Table(dp_inverse_cos, 360);
```

上述调用指定使用 dp_inverse_cos 来存储查找表。查找表中包含 360 个条目，范围为 0～180 度，即精度为 0.5 度。

要使用该查找表，只需调用前面的宏（它假定查找表中有 362 个元素）：

```
FAST_INV_COS(.5)
```

现在，可以重新编写 BSP 插入函数，使之根据观察方向和视点来剔除多边形。

提示：为何要 362 个元素呢？索引为 0～360，因此需要 361 个元素，但如果无需检查越界情况，很多算法的速度都将提高。我们忽略了由于浮点误差导致索引为 361 的情况，然而，如果将第 362 个元素设置为第 361 个元素的值，可避免检查越界情况。

（2）计算观察方向向量

编写新的 BSP 插入函数之前，还有一个问题需要解决：由于没有对世界空间中的几何体执行世界坐标到相机坐标变换，因此不知道观察方向向量。通常，我们使用欧拉角度或 UVN 表示来对世界空间中的几何体进行变换，然后假定相机位于(0，0，0)处，朝向为+z 轴，即观察方向向量为(0，0，1)。

这需要使用一些技巧，因为我们需要的是观察方向向量。然而，在每帧中计算它非常容易，只需使用相机变换矩阵的逆矩阵对向量(0，0，1)进行变换，便可得到观察方向矩阵。然而，变换时不用考虑平移部分，而只需根据旋转角度对向量(0，0，1)进行旋转即可。在欧拉模型中，这些角度为 cam.dir.x、cam.dir.y 和 cam.dir.z。当然旋转顺序至关重要，建立相机变换矩阵时，我们调用的是下述函数：

```
Build_CAM4DV1_Matrix_Euler(&cam, CAM_ROT_SEQ_ZYX);
```

这意味着计算观察方向向量时，需要按 XYZ 的顺序（与上述函数中相反）对向量(0，0，1)进行旋转，且使用非逆变换。换句话说，需要根据旋转角度 cam.dir.x、cam.dir.y 和 cam.dir.z 创建一个 XYZ 旋转矩阵，然后使用该矩阵对向量(0，0，1)进行变换。要创建该变换矩阵，可以这样做：

```
MATRIX4X4 mrot;

Build_XYZ_Rotation_MATRIX4X4(cam.dir.x,
                cam.dir.y,
                cam.dir.z
                &mrot);
```

然后，像下面这样使用上述矩阵对向量(0，0，1)进行变换：

```
VECTOR4D vz = {0,0,1,1}; // z = 1, w = 1

Mat_Mul_VECTOR4D_4X4(&vz,
            &mrot,
            &vdir);
```

现在万事具备了。在使用渲染列表背面消除函数也能很好地完成工作的情况下，这样计算向量和消除背面看似增加了很多计算量，但与浪费时间将额外三角形（每个四边形两个）插入到渲染列表中相比，这样做是绝对值得的，更何况对多余的三角形执行世界坐标到相机坐标变换也会浪费时间。在 BSP 树中，所有的墙面都是四边形，它们还没有被分割成三角形，我们应充分利用这一点。

总之，本节虽然很重要，但主要是为下一节讨论视景体剔除做准备，使用视景体剔除可以立刻删除 BSP 树中的整个子树。具备背面消除功能的 BSP 树插入函数的原型如下：

```
void Bsp_Insertion_Traversal_RemoveBF_RENDERLIST4DV2(
        RENDERLIST4DV2_PTR rend_list, // 渲染列表
        BSPNODEV1_PTR root, // 要插入的 BSP 树的根
        CAM4DV1_PTR cam,   // 相机
        int insert_local=0); // 插入标记
```

该函数的调用方法与前一个版本（不支持背面消除）完全相同。执行背面消除的代码完全遵循了前几页介绍的方法，为节省篇幅，这里没有列出其代码，读者可以在附带光盘中找到它（还有本章的其他所有代码）。

DEMOII13_2.CPP|EXE 是一个使用该函数的演示程序，它是一个标准的关卡编辑器，切换到 3D 模式后，用户可以在屏幕的下方看到 BSP 树插入函数传递到流水线下游的多边形数目。读者可以按 C 键在支持背面消除的 BSP 树插入函数和标准 BSP 插入函数之间切换，并改变视点位置，以查看进入渲染列表的多边形数目的增减情况。

注意：要编译该演示程序，需要 DEMOII13_2.CPP、所有库模块（T3DLIB1-11.CPP|H）以及 DirectX .LIB 文件。另外，还需要用于菜单等的资源文件，该文件名为 DEMOII13_2.RC。最后，请使用 DOOM01.LEV 来试验。

注意：运行该演示程序时，读者可能惊讶地发现，走出房间后，仍然有 BSP 树中的多边形被插入到渲染列表中。这实际上是正确的，因为我们执行的是平面剔除，而不是多边形剔除，如图 13.41 所示。要执行多边形剔除，不但需要检测平面是否可见，还需要检测多边形的端点是否在视景体内，这样做的计算量有些大，可能得不偿失。然而，读者可能想尝试一下。

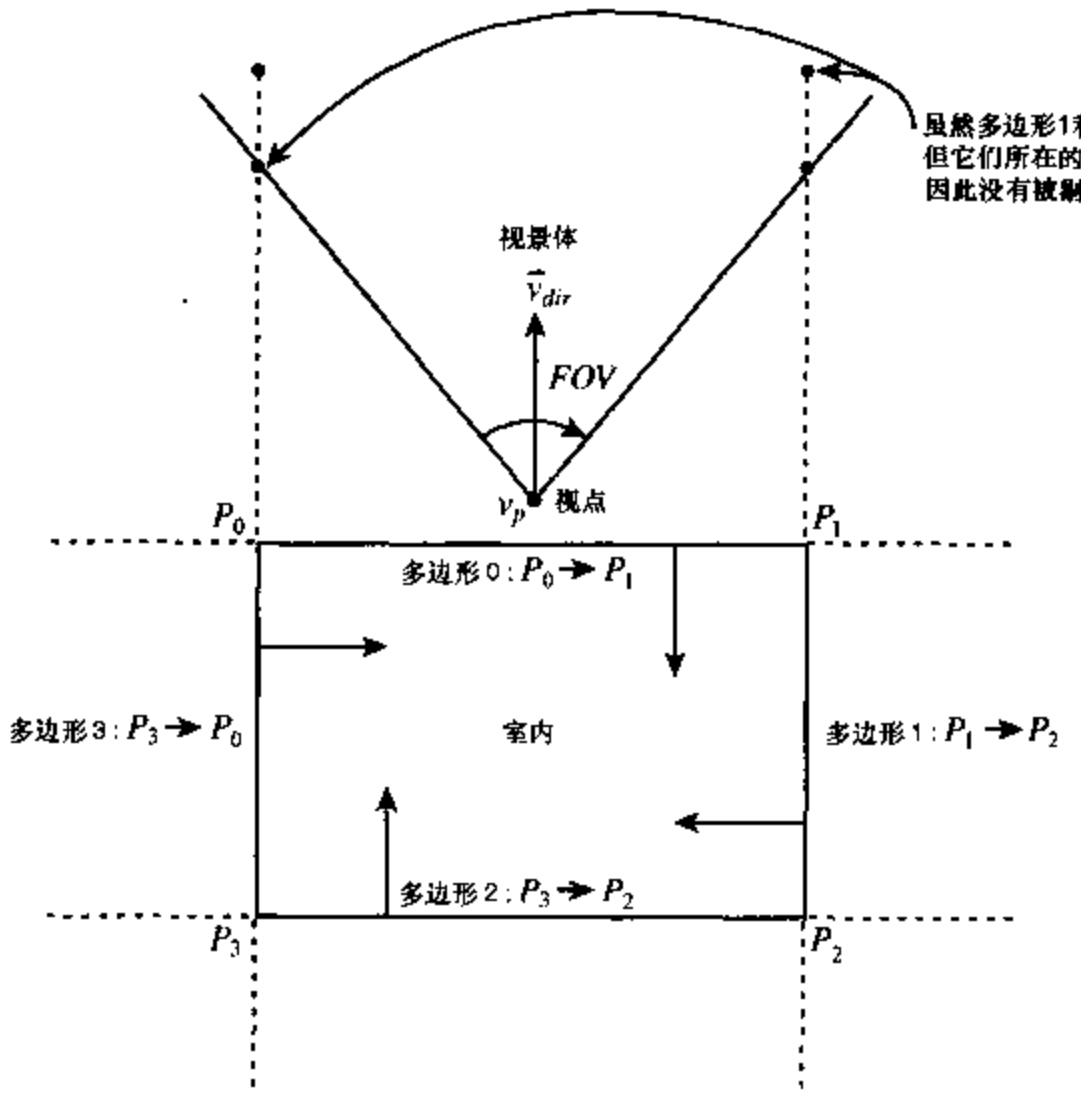


图 13.41　多边形可能是背面，但我们执行的是分割面剔除

2. 视景体剔除

对于 BSP 树，可使用一种比简单背面剔除功能更强大的剔除操作。具有讽刺意义的是，我们在编写背面剔除函数时已经完成了实现这种功能的部分代码。为明白这一点，请看图 13.42。视点位于分割面的负半空间中，由于观察方向向后，分割面是不可见的，更为重要的是，该分割面的整个正半空间面也是不可见的。

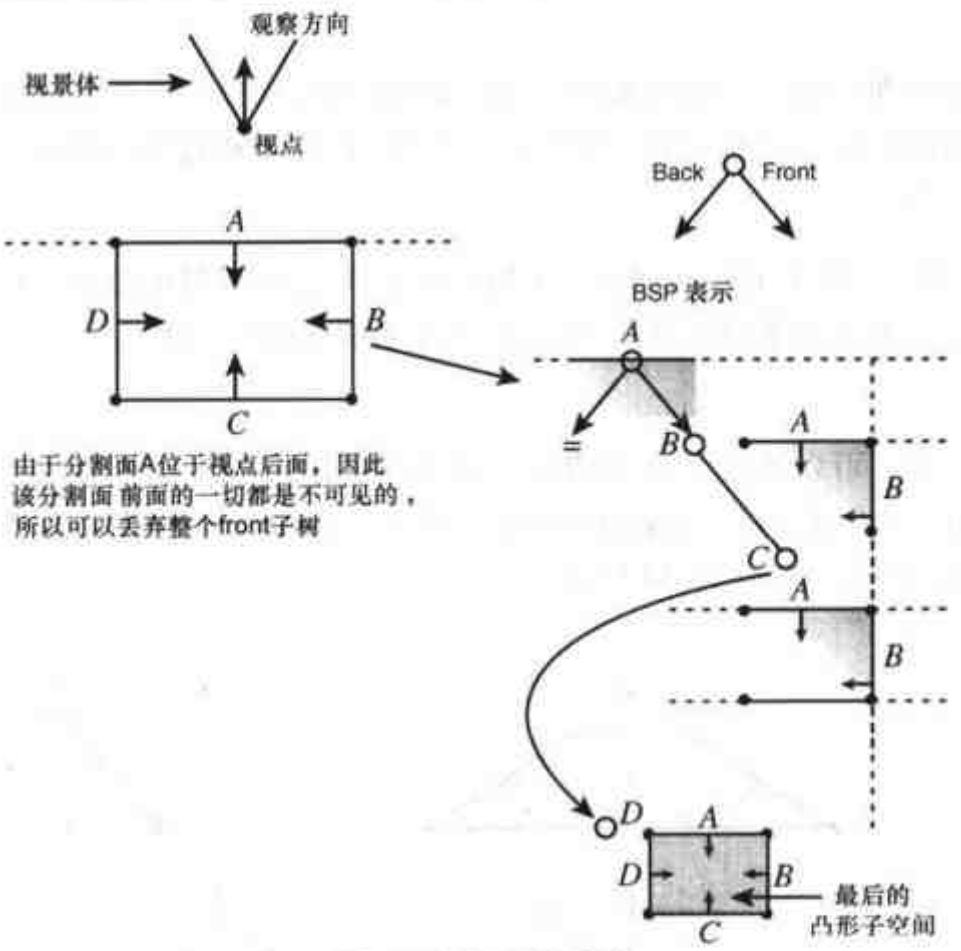


图 13.42　视景体剔除

换句话说，可以丢弃当前节点的整个子树，根本不用考虑它。想象一下功能有多强大吧：给定观察方向，如果从视点看不到前面（或后面）的某个分割面，则不但可以剔除该分割面，还可以剔除它后面（前面）的整个子树。这就是 BSP 树的威力所在，它具有这样的特性：表示的空间被划分为凸形子空间。

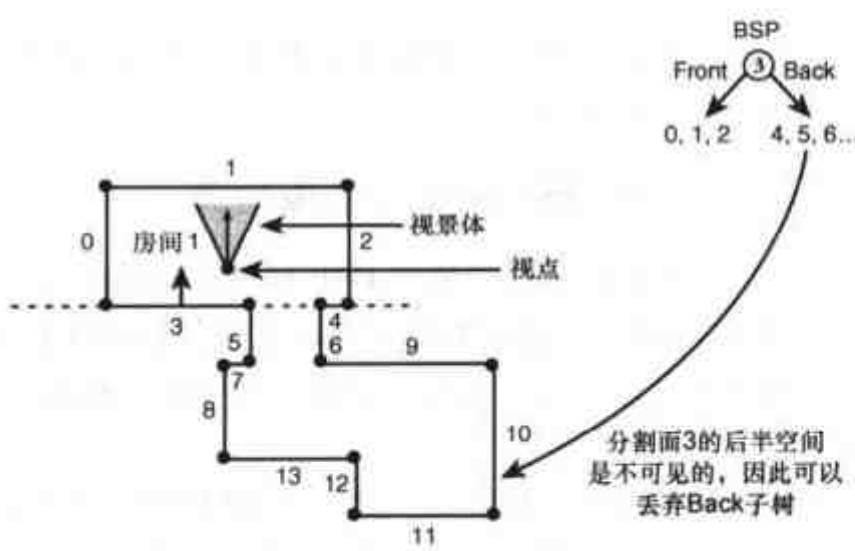


图 13.43　剔除整个房间

为进一步让读者相信这一点，请看图 13.43。这是一个包含两个房间的简单范例（当然，假设已经创建了 BSP 树），视点位于房间 1 中，玩家后面的墙面和房间 2 都不可见。发现墙面 3 不可见后，可以立刻确定它后面的整个子空间都不可见，因此无需沿 BSP 树往下遍历。

实现视景体剔除非常简单——我们已经完成其中部分工作。可以编写一个独立的函数，对被剔除的节点及其子节点进行标记；也可以像背面剔除函数那样，在将 BSP 树插入到渲染列表中的同时执行视景体剔除。实在没有必要遍历 BSP 树两次，我们将把视景体剔除功能和 BSP 树插入操作集成到一个函数中。方法是，将访问 front 或 back 子树的递归调用封装到剔除操作中，换句话说，确定多边形被剔除后，便知道它后面的一切也将被剔除，因此无需通过递归调用函数来遍历 back 子树。这只需修改一行代码。

实现这种功能的函数的原型如下：

```
void Bsp_Insertion_Traversal_FrustrumCull_RENDERLIST4DV2(RENDERLIST4DV2_PTR
 rend_list,
                                BSPNODEV1_PTR root,
                                CAM4DV1_PTR cam,
                                int insert_local=0);
```

该函数的调用方法与前两个函数相同，惟一的差别是，它执行背面剔除和视景体剔除。

DEMOII13_3.CPP|EXE 是一个使用该函数的演示程序。它是一个标准的关卡编辑器，切换到 3D 模式后，用户可以在屏幕的下方看到 BSP 树插入函数传递到流水线下游的多边形数目。读者可以按 C 键在支持视景体剔除的 BSP 树插入函数和标准 BSP 插入函数之间切换，并改变视点位置，以查看进入渲染列表的

多边形数目的增减情况。

读者将发现，使用支持视景体剔除的函数时，进入渲染列表中的多边形将比使用支持背面剔除的函数时更少，因为整个子树都被剔除，这样在该子树中，其所在平面可见的多边形将不再像使用简单背面剔除函数时那样，成为漏网之鱼。

注意：要编译该演示程序，需要 DEMOII13_2.CPP、所有库模块（T3DLIB1-11.CPP|H）以及 DirectX .LIB 文件。另外，还需要用于菜单等的资源文件，该文件名为 DEMOII13_2.RC。最后，请使用 DOOM01.LEV 来试验。

最后需要指出的一点是，可以通过判断多边形（而不是其所在的平面）是否在视景体内，使用有限的视景体并启用视景体裁剪，来使视景体剔除的功能更为强大。这样，确定多边形在近（或远）裁剪面之外后，便可以丢弃整个 back 子树，如图 13.44 所示。

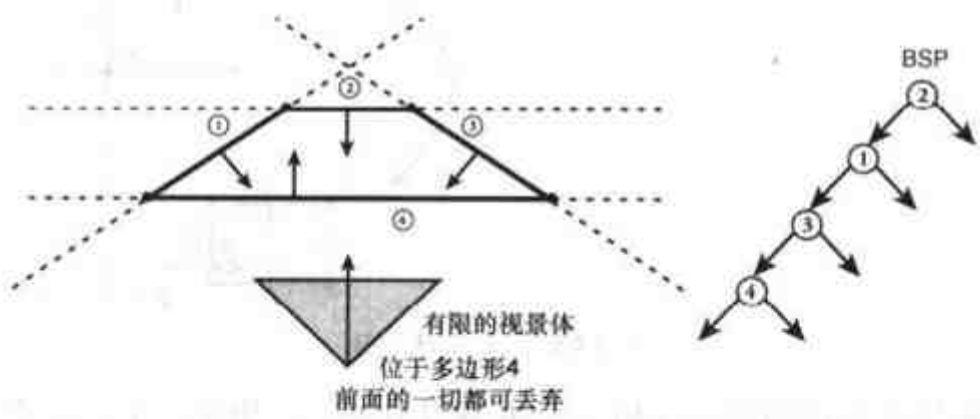


图 13.44　使用有限视景体实现更为积极的视景体剔除

至此，读者应该明白，BSP 树提供了无限的可能性，它确实有局限性，但如果能将其用于系统中，它们将是强大的“同盟国”。

13.3.14　将 BSP 树用于碰撞检测

最后，简要地介绍一下使用 BSP 树进行碰撞检测（本书后面讨论碰撞检测时，将更深入地探讨这个主题）。BSP 树可用于快速判断是否与玩家或其他物体发生碰撞。在大多数情况下，使用包围体（如立方体、球体或圆柱体）来包围玩家或非玩家角色网格，但确定碰撞面后，使用 BSP 树来快速判断是否真的发生了碰撞。

BSP 树的优点是，您能够快速地沿 BSP 树遍历 3D 空间，直到找到一个这样的叶节点：物体位于其凸形子空间中。这种操作平均只需 $\log_2^n$ 就能完成，当然，如果 BSP 树是完全不平衡的，将需要 n 步。这是因为，每次迭代时，通过将视点或物体的中心同节点进行比较，便可以排除 3D 空间的一半，直到找到满足条件的叶节点。

找到满足条件的叶节点后，便可以根据 BSP 节点的几何形状来检测是否已经（或将要）发生碰撞，方法是扩大包围体或沿法线向量移动 BSP 节点（沿法线方向将分割面往前推），直到发生碰撞为止。

13.3.15　集成 BSP 树和标准渲染

仅当读者需要遍历 BSP 树以确定准确的从后到前（或从前到后）的渲染顺序时，本节才显得重要。如果读者对此不感兴趣，这种集成将无关紧要；如果读者对此感兴趣，请继续往下阅读。

1．与 z 排序的集成

集成 BSP 树和 z 排序比较棘手，因为 BSP 树中的多边形排列顺序已经是正确的，而 z 排序将重新排列多边形的顺序。然而，在 BSP 关卡编辑器演示程序中，作者采取了类似的做法。作者知道，地面和天花板总是在墙面的“下面”，因此可以像画家那样，先渲染它们，然后在按 BSP 树中的顺序渲染墙面。对于简单的分层场景，可以对背景和前景进行 z 排序，条件是不对 BSP 树中的多边形重新排序。

例如，在 BSP 演示程序中，地面多边形都是共面的，因此渲染顺序无关紧要，因为它们之间不可能相互

遮掩。然而，如果地面不是平的，即凸起的地方，则必须首先渲染没有凸块的多边形，然后渲染有凸块的多边形。在这种情况下，仍可以使用混合方法来处理：加载包含所有地面多边形的渲染列表，对其进行 z 排序，然后插入 BSP 树。只要 BSP 树中的几何体不比地面上的几何体小（即不存在这样的情况，BSP 树中有一个很小的墙面，可能被地面上的东西遮掩），对渲染列表进行渲染时，便可得到正确的从后到前的顺序。

就个人而言，作者不喜欢结合使用 z 排序和 BSP 树，因为除地面、天花板等已经被标记为位于 BSP 树前面或后面的几何体外，这几乎毫无用处。z 缓存是一种更好的方法。

2. 与 z 缓存的集成

BSP 树渲染旨在确定正确的从后到前的渲染顺序（可能结合使用 PVS 来最大限度地减少多边形重绘），然后在不使用 z 缓存的情况下来绘制多边形。然而，对于可移动的物体，这种方法不管用，因为不可能在每次物体移动后都重新生成 BSP 树（对于复杂的游戏关卡，这需要几秒、几分甚至几小时）。

注意：多边形在其所在的平面中移动，且不影响其他 BSP 节点表示的凸形子空间时，无需重新生成 BSP 树。这种特性对于门、电梯等很有用。

我们必须找到一种集成 BSP 树渲染和 z 缓存的方法。如果对 BSP 树使用 z 缓存，将有悖于 BSP 渲染的初衷，它旨在通过顺序遍历获得正确的从后到前的渲染顺序。技巧如下：创建特殊的光栅化函数，在其中不对像素的 z 值同 z 缓存进行比较，而总是将像素的 z 值写入到 z 缓存中。也就是说，绘制每个像素时，不执行读取-比较-写入 z 缓存的操作，而只执行写入 z 缓存的操作（直接写入，write-through）。这样便得到了一个 z 缓存，供下一个阶段使用。这样做之所以能够得到正确的 z 缓存，是因为遍历 BSP 树时，将按从后到前的顺序访问多边形，从而按从后到前的顺序根据多边形更新 z 缓存。图 13.45 说明了这一点。

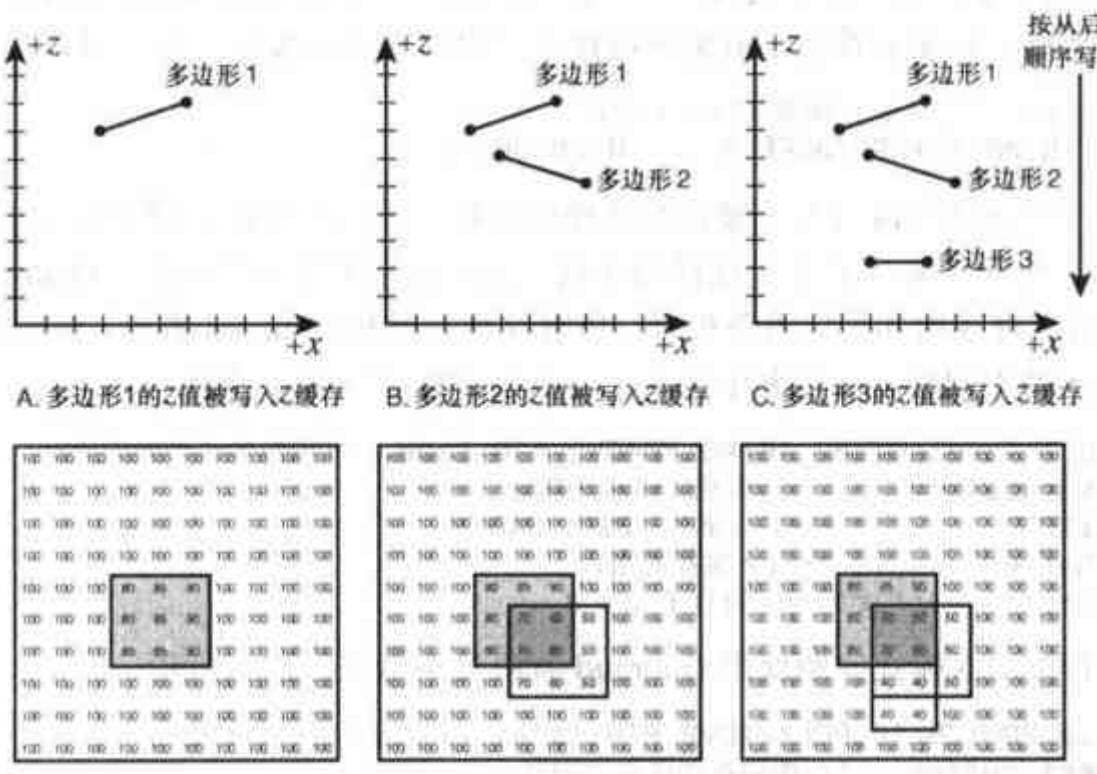


图 13.45　按从后到前的顺序遍历多边形，从而用最小的 z 值更新 z 缓存

例如，下面的代码摘自函数 Draw_Triangle_2DZB_16()，该函数使用 z 缓存绘制采用恒定着色的三角形，但不具备直接写入功能：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
  // 检测像素的 z 值是否小于 z 缓存中相应的值
  if (zi < z_ptr[xi])
    {
    // 绘制像素
     screen_ptr[xi] = color;

    // 更新 z 缓存
    z_ptr[xi] = zi;
```

```
    } // end if

   // 计算下一个像素的 z 值
   zi+=dz;
   } // end for xi
```

删除 z 值比较功能，并加入直接写入功能后的代码如下：

```
// 绘制扫描线
for (xi=xstart; xi<=xend; xi++)
  {
  // 绘制像素
  screen_ptr[xi] = color;

  // 更新 z 缓存
  z_ptr[xi] = zi;


  // 计算下一个像素的 z 值
  zi+=dz;
  } // end for xi
```

现在，可以使用 z 缓存来渲染移动的物体了。在这个阶段，我们将使用标准的读取-比较-写入算法，但不清除 z 缓存。我们使用渲染 BSP 树中的多边形时生成的 z 缓存来绘制物体。这样，既使用了 BSP 树来确保墙面的正确渲染顺序，同时支持移动的物体，它们可以移动到墙面后，与墙面相互贯穿等。

直接写入 z 缓存的光栅化函数

为支持 z 缓存直接写入功能，需要将其加入到光栅化函数中（只编写了几个这样的函数，供演示程序使用）；定义一个渲染场境常量，用于指定直接写入 z 缓存；编写一个新的渲染场境函数，以正确地调用直接写入 z 缓存的光栅化函数。首先来看指定渲染场境模式“直接写入 z 缓存”的常量：

```
// 使用 z 缓存，但不比较 z 值，而是直接将其写入 z 缓存
#define RENDER_ATTR_WRITETHRUZBUFFER      0x00000008
```

接下来，需要编写一些支持直接写入 z 缓存的光栅化函数。这只需删除 z 缓存检测逻辑，直接将像素的 z 值写入 z 缓存即可。作者只编写了 5 个这样的函数，它们分别支持恒定着色、Gouraud 着色、仿射纹理映射和固定着色、仿射纹理映射和恒定着色处理、仿射纹理映射和 Gouraud 着色。这些函数的原型如下，与前一个版本相比，惟一的不同是在函数名中加入了字符串“WT”（表示直接写入）：

```
void Draw_Gouraud_TriangleWTZB2_16(POLYF4DV2_PTR face, //指向多边形面的指针
       UCHAR *_dest_buffer,   // 指向视频缓存的指针
       int mem_pitch,      // 每行多少字节（320、640 等）
       UCHAR *_zbuffer,     // 指向 z 缓存的指针
       int zpitch);       // z 缓存中每行多少字节
```

该函数使用 z 缓存直接写入模式绘制使用 Gouraud 着色且不带纹理的三角形。

```
void Draw_Triangle_2DWTZB_16(POLYF4DV2_PTR face, //指向多边形面的指针
       UCHAR *_dest_buffer,  // 指向视频缓存的指针
       int mem_pitch,      // 每行多少字节（320、640 等）
       UCHAR *_zbuffer,     // 指向 z 缓存的指针
       int zpitch);       // z 缓存中每行多少字节
```

该函数使用 z 缓存直接写入模式绘制使用固定/恒定着色且不带纹理的三角形。

```
void Draw_Textured_TriangleGSWTZB_16(POLYF4DV2_PTR face, //指向多边形面的指针
           UCHAR *_dest_buffer,  // 指向视频缓存的指针
           int mem_pitch,      // 每行多少字节（320、640 等）
           UCHAR *_zbuffer,     // 指向 z 缓存的指针
           int zpitch);       // z 缓存中每行多少字节
```

该函数使用 z 缓存直接写入模式和仿射纹理映射绘制使用 Gouraud 着色的三角形。

```
void Draw_Textured_TriangleFSWTZB2_16(POLYF4DV2_PTR face, //指向多边形面的指针
           UCHAR *_dest_buffer,   // 指向视频缓存的指针
           int mem_pitch,      // 每行多少字节（320、640 等）
```

```
            UCHAR *_zbuffer,     // 指向 z 缓存的指针
            int zpitch);       // z 缓存中每行多少字节
```

该函数使用 z 缓存直接写入模式和仿射纹理映射绘制使用恒定着色的三角形。

```
void Draw_Textured_TriangleWTZB2_16(POLYF4DV2_PTR face, //指向多边形面的指针
            UCHAR *_dest_buffer,    // 指向视频缓存的指针
            int mem_pitch,       // 每行多少字节（320、640 等）
            UCHAR *_zbuffer,      // 指向 z 缓存的指针
            int zpitch);       // z 缓存中每行多少字节
```

该函数使用 z 缓存直接写入模式和仿射纹理映射绘制使用固定着色的三角形。

除了不检测 z 值，而是直接将其写入 z 缓存外，这些函数与前一个版本相同。用户必须确保在使用 z 缓存渲染可移动的物体之前，以从后到前的顺序渲染 BSP 树中的多边形。

最后，需要一个新的渲染场境函数，它包含一个处理下述 z 缓存模式的代码块：

```
RENDER_ATTR_WRITETHRUZBUFFER
```

这个函数的原型如下：

```
void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_2(RENDERCONTEXTV1_PTR rc);
```

除支持新的 z 缓存直接写入模式以及知道如何调用直接写入 z 缓存的光栅化函数外，它与前一个版本 Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16() 相同。

最后，下面是一个设置渲染场境，以直接写入 z 缓存模式绘制多边形的例子：

```
// 设置渲染场境，使用直接写入 z 缓存模式
rc.attr = RENDER_ATTR_WRITETHRUZBUFFER |
     RENDER_ATTR_TEXTURE_PERSPECTIVE_AFFINE;
// 清除 z 缓存
Clear_Zbuffer(&zbuffer, (32000 << FIXP16_SHIFT));

rc.video_buffer  = back_buffer;
rc.lpitch     = back_lpitch;
rc.mip_dist    = 0;
rc.zbuffer    = (UCHAR *)zbuffer.zbuffer;
rc.zpitch     = WINDOW_WIDTH*4;
rc.rend_list   = &rend_list;
rc.texture_dist  = 0;
rc.alpha_override = -1;

// 对场景进行渲染
Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_2(&rc);
```

除调用的是函数 Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_2() 外，与以前惟一不同是使用了标记 RENDER_ATTR_WRITETHRUZBUFFER，而不是标记 RENDER_ATTR_ZBUFFER。然而，在渲染静止的多边形之前，仍需要清除 z 缓存。在这次渲染中，将直接把 z 值写入 z 缓存，但仍需要初始化 z 缓存；否则，接下来使用 z 缓存来渲染移动的物体时，结果将不正确。

结合使用 BSP 渲染和 z 缓存的步骤如下：

（1）初始化 z 缓存。

（2）使用 z 缓存直接写入模式渲染所有的静止几何体。这次的渲染顺序必须是从后到前的。

（3）使用 z 缓存渲染移动的几何体。

（4）显示帧。

3. 移动物体范例

为编写演示移动物体和 BSP 的程序，作者对关卡编辑器演示程序进行修改，以加载一个 Tron 识别器，并使之在游戏世界中沿圆形轨迹移动。移动方向总是平行于圆形轨迹的切线，如图 13.46 所示。

修改的地方包括：新增了对 z 缓存直接写入模式的支持；在 Game_Main() 中加入了移动物体的代码。这样做旨在向读者证明，这种技术确实管用。该演示程序名为 DEMOII13_4.CPP|EXE，图 13.47a 和 13.47b 是其运行时的屏幕截图（分别为线框模式和实心模式）。它与关卡编辑器演示程序基本相同，但用户通过选

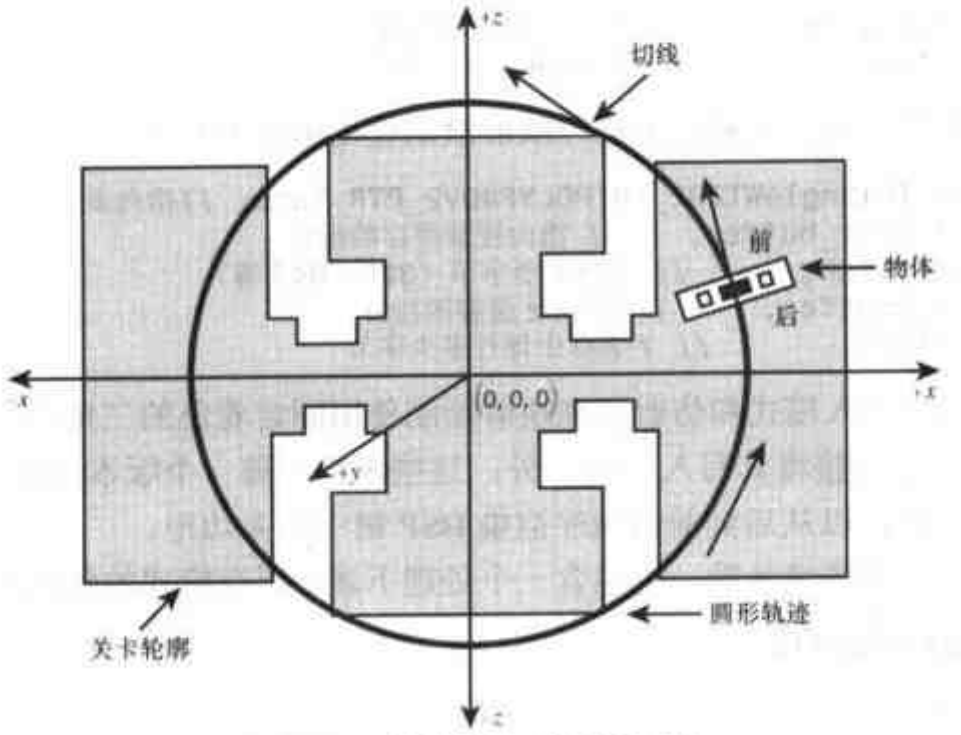


图 13.46　物体的移动轨迹和方向

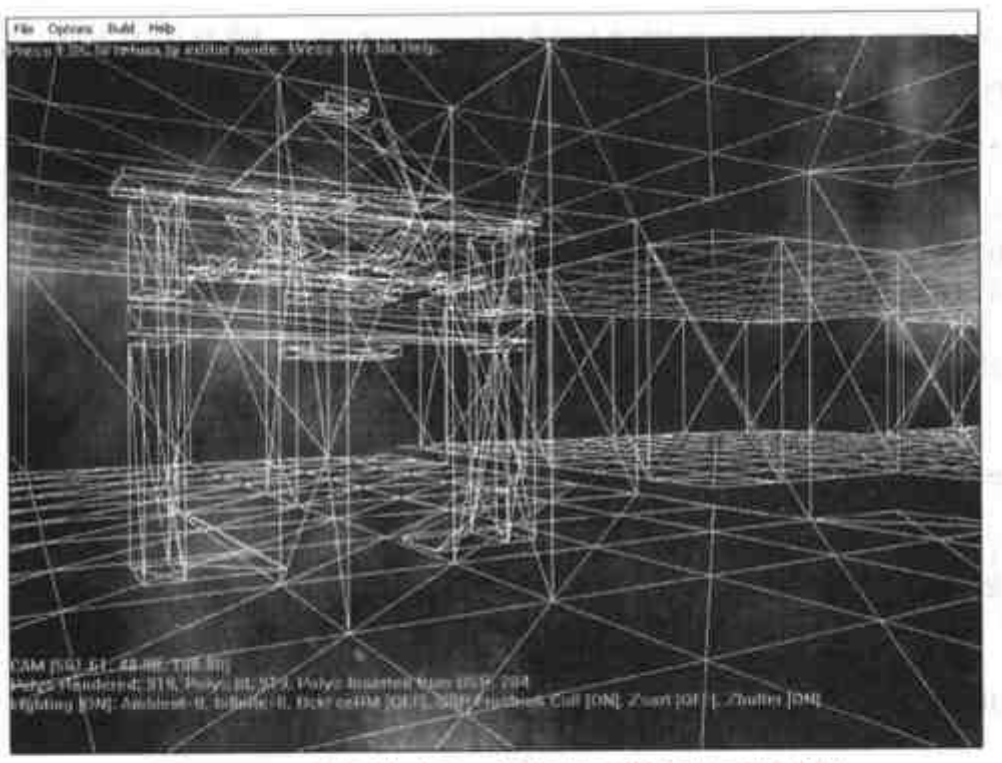


图 13.47a　线框模式下 z 缓存演示程序的屏幕截图

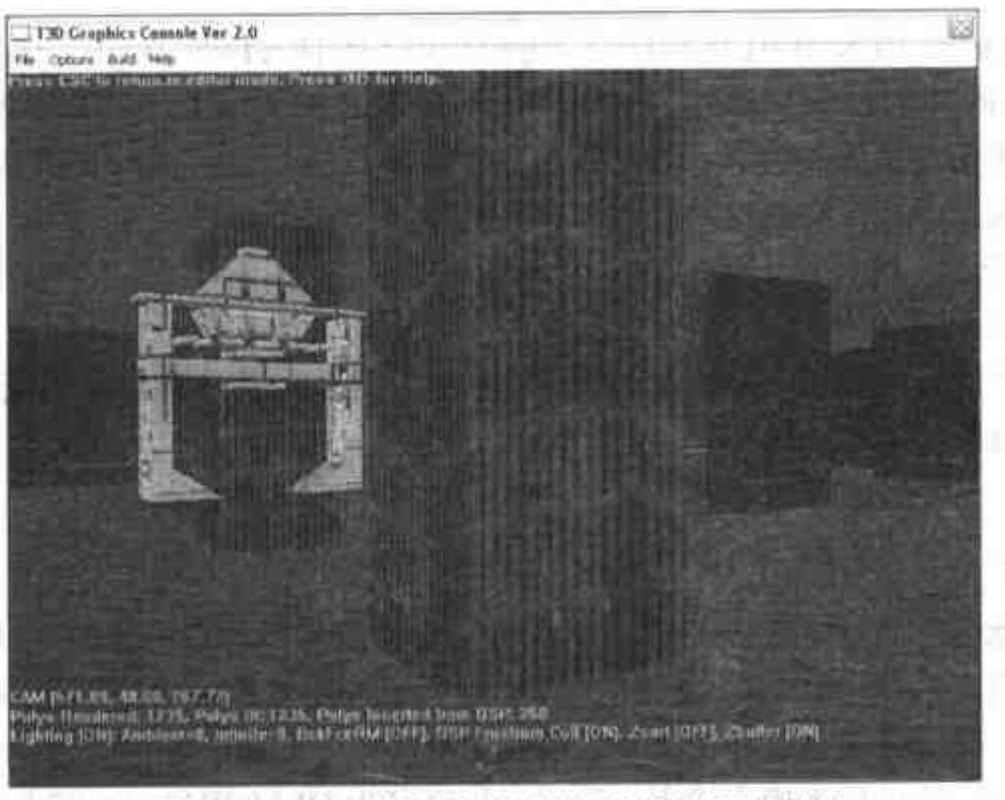


图 13.47b　实心模式下 z 缓存演示程序的屏幕截图

择菜单 Build/Compile BSP and View 进入 3D 模式后，将有一个物体在关卡中移动，该物体是使用 z 缓存渲染的。读者可以加载自己创建的任何关卡，如果没有，可以使用 DOOM01.LEV。要运行该演示程序，首先需要选择菜单 File/Load .LEV File 来加载一个关卡，然后选择 Build/Compile BSP and View 切换到 3D 模式。另外，还可以按 z 键来开/关 z 缓存。

注意：要编译该演示程序，需要 DEMOII13_4.CPP、所有库模块（T3DLIB1-11.CPP|H）以及 DirectX .LIB 文件。另外，还需要用于菜单等的资源文件，该文件名为 DEMOII13_2.RC。最后，请使用 DOOM01.LEV 来试验。

13.4　潜在可见集

我们一直在重复的一个论点是，避免处理那些从视点处看不到（位于视景体外）的几何体是件好事。前面介绍过，使用 BSP 树可以大规模地剔除 BSP 节点，因为如果 BSP 节点不在视景体内，其后面的一切也不在视景体内，可以简单地将它们剔除。这样，将可以删除大量的几何体。

然而，可见的几何体将留下。如果视景体内有 10 000 个多边形，即使使用零重绘技术，仍需要处理那些不需要绘制的多边形。如果一开始就能够确定不需要绘制它们将如何呢？也就是说，给定任何视点，都有一个从该视点可能能够看到的多边形列表将如何呢？可以对大型几何体执行剔除操作，得到位于视景体内的多变形列表，然后以视点作为索引找到潜在可见集（potentially visible set，PVS），只绘制同时位于这两个列表中的多边形。

实现这种理念的方法很多——使用 BSP、八叉树或什么也不用，因此这里将花些时间来讨论它。假设读者使用关卡编辑器创建了如图 13.48 所示的房间。

在这个图中，有 6 个与坐标轴平行的房间，每个房间都由多个多边形构成，房间中可能还有由多边形构成的物体。出于简化的目的，假设网格数据是静态的。现在来创建房间 1 的 PVS。这对应的问题是，在房间 1 中可能能够看到哪些房间？我们可以立刻判断出：在房间 1 中，通过房门（doorway）可以看到房间 1、2、3 和 5。

注意：这里将房间 1 包含在内是因为，视点所在房间内的多边形总是可能可见的。

如图 13.48b 所示，可以将这些房间存储在一个数据结构中，每个房间都有一个列表，其中包含从该房间可能能够看到的所有房间。当然，也可以使用一个 m×m 的邻接矩阵（adjacency matrix）来表示，其中 m 为房间数，矩阵元素(i，j)（i 为行，j 为列）的值表示从房间 i 到房间 j 的可见性。例如，图 13.49 是这个例子的房间邻接图。

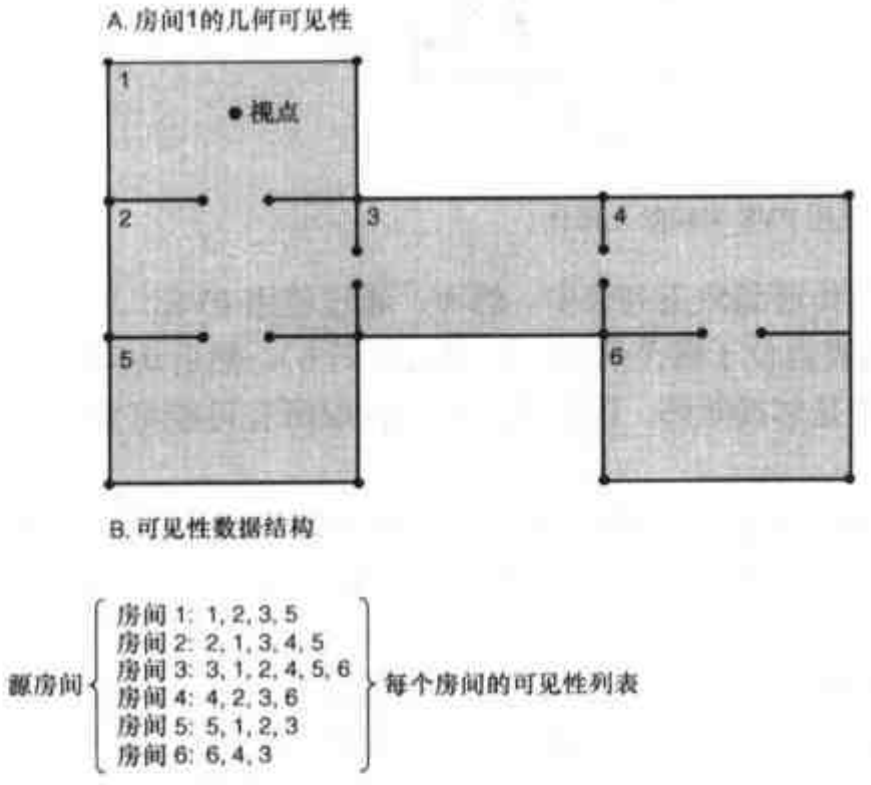


图 13.48　房间到房间的可见性

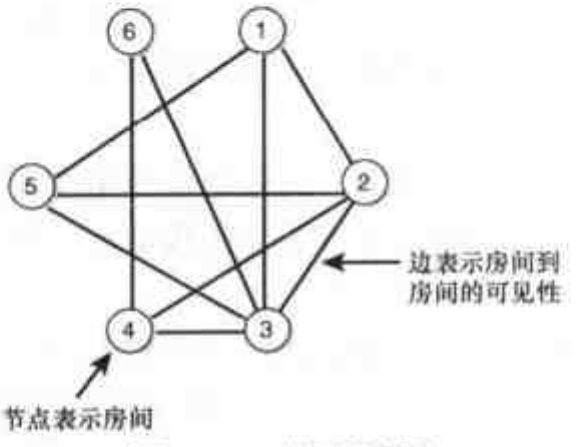


图 13.49　房间邻接图

下面是该关卡的邻接矩阵表示：

```
   i1 i2 i3 i4 i5 i6
j1  1  1  1  0  1  0
j2  1  1  1  1  1  0
j3  1  1  1  1  1  1
j4  0  1  1  1  0  1
j5  1  1  1  0  1  0
j6  0  0  1  1  0  1
```

从中可知，从任何房间看，关卡中的大部分房间都是可能可见的。从房间 6 中看，可能看到的房间数最少。现在，PVS 的“粒度”为房间，换句话说，可能可见的房间列表中包括视点所在的房间。这很好，我们暂时将这个话题搁下，先来讨论一下如何使用 PVS。

13.4.1 使用潜在可见集

假设读者创建了一个关卡，它包含一组房间，并创建了每个房间的 PVS，利用 PVS 来减少重绘很容易。例如，假设读者使用 BSP 树或其他空间分割系统来表示各个房间。根据视景体进行了剔除操作，包括使用 BSP 剔除删除了环境中大部分多边形。现在，可以将多边形传递到流水线的下游，进行裁剪、光照处理、投影、背面消除等。现在是 PVS 发挥作用的时候了。您首先判断视点位于哪个房间中（这可以使用 BSP 来完成），或者将游戏关卡进行分区（sectorizing），进而判断视点位于哪个单元格（cell）中。假设视点位于房间 6 中，视景体如图 13.50 所示。

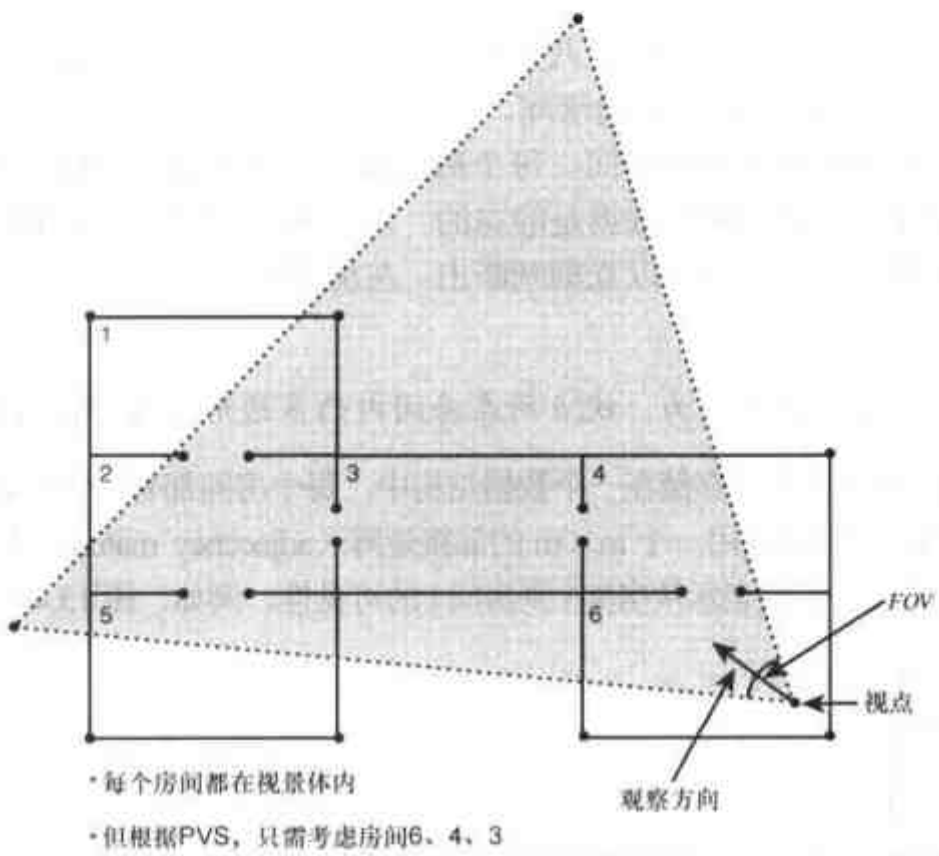


图 13.50 使用 PVS 来剔除几何体

即使使用 BSP 剔除，这个模型数据库也可能传递到渲染列表中。然而，通过使用 PVS，几乎可以完全避免重绘。这简单得令人难以置信：您只需确定视点位于哪个房间（这里为房间 6），然后访问包含该房间的 PVS 的数据结构。在这个例子中，我们使用的是邻接矩阵，因此从第 6 列读取所有可能可见的房间：房间 3、4 和 6。

令人惊讶的是，从视景体的角度看，每个房间都是可见的，因此几乎所有的多边形都将传递到流水线中。有些多边形确实会被裁剪掉，但很多完全被房间 6 的墙面遮掩的多边形将被绘制——至少会检测其 z 值。

这是完全不能接受的，如果您的关卡类似于图 13.51（这是一个简单的 Doom 关卡），包含大量的房间，重绘量将非常大——屏幕将被重绘很多次。PVS 是一个不可思议的工具，可以提高渲染速度，几乎可以避免重绘。

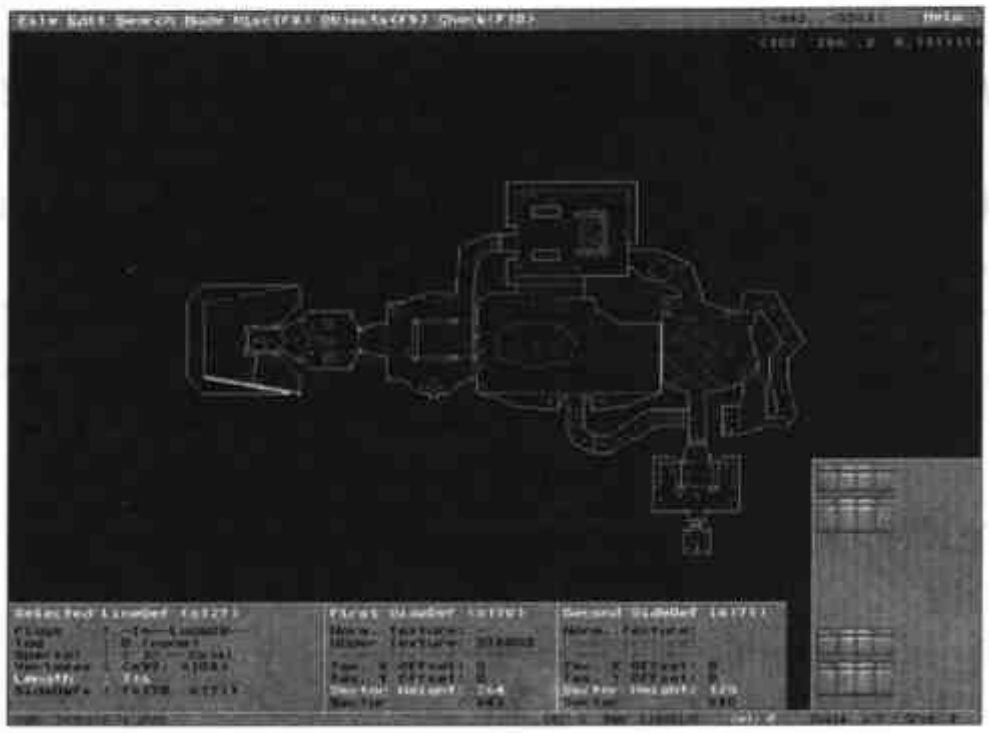


图 13.51　即使是简单 Doom 关卡也可能带来重绘恶梦

有些多边形被部分遮掩，但仍然被包含在 PVS 中。这是完全可以接受的，从作者的经验看，使用前面介绍的 PVS 时，重绘比例最多为 50～150%；但如果不使用 PVS，对于大型关卡，重绘比例很容易达到 1000～2000%。也就是说，仅仅通过使用 PVS，就可以将帧频（多边形数量）提高（降低）一个数量级。

13.4.2　潜在可见集的其他编码方法

当关卡很大时（包含 10 000～1 000 000 个多边形），PVS 将非常大。在我们的简单范例中，使用了两种方法：每个房间一个相关联的数组或邻接矩阵。对于房间到房间的 PVS（指出房间的可见性，而不是多边形的可见性）而言，这两种技术都是可行的。

如果要对从任何视点看，多边形的可见性信息进行编码，可使用分区技术，根据视点所在的区段（sector），计算每个多边形的可见性。找出可见的多边形是一个几何问题，稍后再介绍，但通常是基于光线和视线（line-of-sight）可见性的，因此并不难。接下来花些时间介绍如何对大型数据进行编码。

假设关卡中有 50 000 个多边形和 100 个房间，则有两种方法可供选择。可以根据视点所在的房间来进行 PVS 编码；也可以将游戏世界划分成小型正方形或立方体，然后为每个正方形或立方体，创建一个可能可见的多边形列表。图 13.52 概括地说明了这一点。

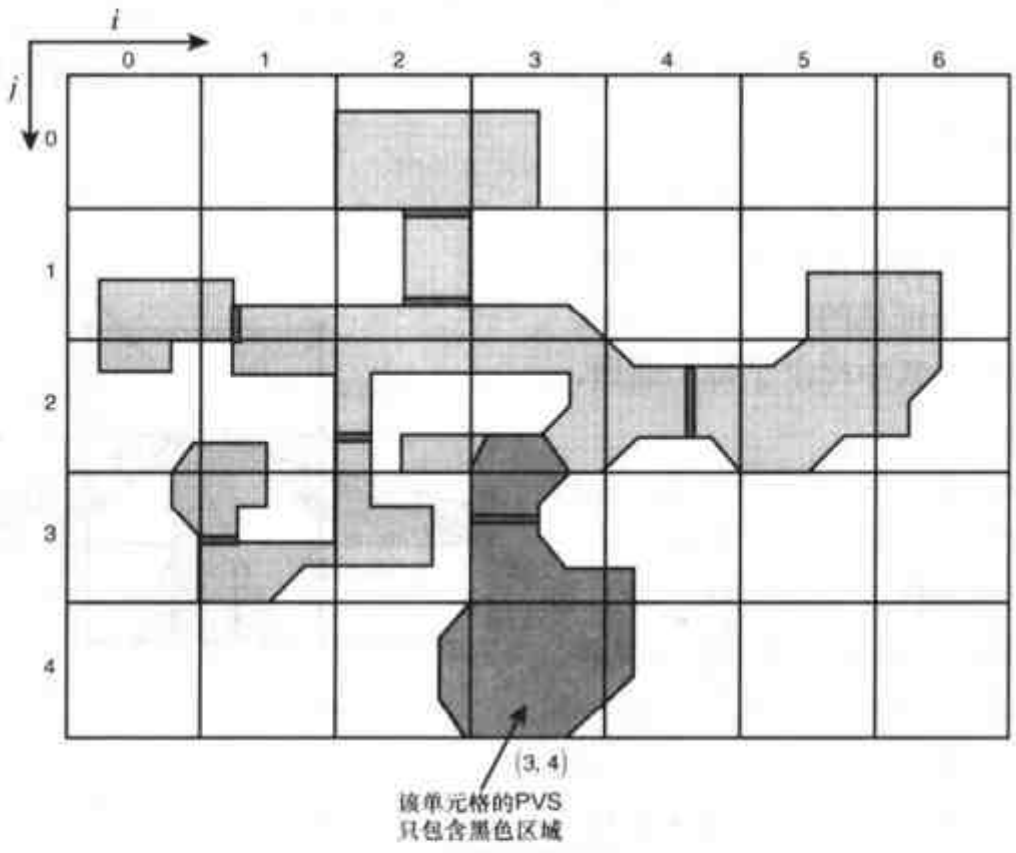


图 13.52　基于分区的 PVS

如果使用索引来进行 PVS 编码，则对于给定房间或分区，每个可能可见的多边形都需要一个索引。例如，在前面包含 100 个房间（有 50 000 个多边形）的例子中，从每个房间看，平均大约有 1000 个多边形是可能可见的。这意味着 PVS 的大小为：100 个房间×1000 个可能可见的多边形×4 字节 =400K 字节。

这实际上并不太糟糕。当然，还可以做得更好些：使用 3 字节的索引，而不是使用 4 字节（32 位）的整数索引。在另一方面，如果用另一种方式进行 PVS 编码，效率可能更高些。

例如，可以使用一个位表（bit list），其中的各位依次对应于多边形 0～*n*，如果多边形是可能可见的，则将相应的位设置为 1。换句话说，如果有 10 个多边形，从房间 *n* 可以看到多边形 0、5 和 6，则房间 *n* 的 PVS 如下：

1000011000

当然，使用这种位编码时，必须包含所有多边形的可见性值，但确实可以节省内存。使用这种编码方法时，需要的内存量如下：

100 个房间×50 000 个多边形×1 位 =625K 字节。

前面不是说需要的内存量更少吗？确实如此，条件是对其进行压缩。在大多数情况下，使用简单的串长（run-length）编码方法（将连续的 0 或 1 进行压缩）可以将 PVS 压缩 1～20 倍，因此，在大多数情况下，实际所需的存储空间大约为 50KB。

总之，如何存储 PVS——使用多边形索引还是位表——取决于您。这两种方法各有利弊，就个人而言，作者不想节省内存，因此喜欢使用多边形索引。

13.4.3 流行的 PVS 计算方法

前面我们一直避开了计算 PVS 的细节，并非因为它很难，而是因为没有放之四海皆准的方法。计算 PVS 的方法很多，这里只介绍两种最流行的技术。

1. 基于工具的 PVS

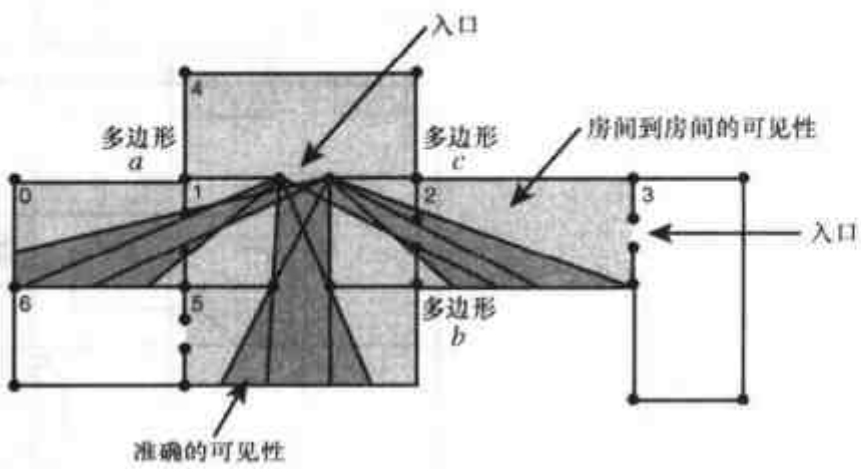
最简单的 PVS 创建方法是手工来完成。例如，假设您创建了一个工具，用它来绘制 2D 世界，然后通过延展（extrude）来得到 3D 世界。每个房间都由一组多边形组成，建立 2D 世界模型后，您进入下一个阶段：手工创建邻接矩阵或其他支持数据结构，用于指出从各个房间看，哪些房间是可见的。这种工作本可以通过直线绘制、泛填充（flood filling）来完成，但为何将问题复杂化呢？如果游戏世界很小，只有 10～50 个房间，可以通过观察，快速地确定每个房间的邻接信息，而无需使用复杂的算法。

推荐读者一开始使用这种方法。手工或使用工具来创建 PVS，以便能够使用 PVS 来完成工作。由于 PVS 通常是离线创建的，因此时间不是问题。然而，如果需要不断修改游戏世界，则不断地重新创建 PVS 将是令人痛苦的。PVS 只是一种优化方法，可以等到对游戏世界完全满意后再创建 PVS。

2. 基于入口/光线的 PVS

一种更常见的 PVS 生成方法是，使用单元格到单元格的入口可见性或光线的特性来找出从任何房间、分区或单元格看，可能可见的多边形。例如，请看图 13.53 所示的关卡，其中标出了每个通道，它们被称为入口（portal）。

另外，每个房间都由独立的多边形组成，我们将创建包含多边形的 PVS。从图 13.53 可知，从房间 4 看，房间到房间的可见性为灰色区域。我们要计算更精确的可见性，即标记各个多边形的可见性。为此，可计算从房间 4 出发、到达其他房间的各条光线。这使用手工方式几乎是不可能完成的，但如果将房间 4 的入口视为光源，将其他入口视为遮光板，进而跟踪光锥，将找到所需的东西：图 13.53



多边形 *a*、*b*、*c* 包含在基于房间的PVS中，
但不在基于多边形的PVS中

图 13.53 基于多边形的入口可见性

所示的黑色区域。

正如读者看到的，使用光线跟踪，确定了从房间 4 的入口出发，经邻接房间进入其他潜在可视房间的光锥，被光线照射到的锥形区域（体）是从房间 4 可能可见的。任何其边位于该区域（体）中的多边形都将包含在房间 4 的 PVS 中。另外，还必须包含房间 1 中的所有多边形，因为它与房间 4 相邻，入口边界与它之间没有其他房间。

使用这种新技术，可避免房间 4 的 PVS 包含房间 0、1、2 和 5 中的所有多边形，而只包含这样的多边形：位于以房间 4 的入口为光源的光锥内或与之相交。在这个例子中，房间 4 的 PVS 包含的多边形数量将减少 30～50%。例如，多边形 a、b 和 c 不包含在 PVS 中，虽然它们包含在房间到房间的 PVS 中。

这种技术也适用于房间中的物体。物体是由多边形组成的，可以检查它们是否在光锥内。当然，这只适用于静态物体，如果房间中的物体移动，可能变得不可见，但仍将包含在 PVS 中。然而，可以通过创建“阴影”区域，确定一些区段（sector），仅当物体在这些区段中时，才将其包含到 PVS 中。物体移动到区段外后，将被遮住。

正如读者看到的，可以使用 PVS 概念来进行各种优化、处理特殊情况等——这只受限于您的想象力。PVS 的计算是离线完成的，因此，可以根据需要花任意长的时间来确定从给定视点或房间看，哪些多边形是可见的。

如果游戏世界不是基于房间的，没有可将其视为光源（遮光板）的入口（门），该如何办呢？这不是问题，只是需要一些创造力。一种可行的技巧是，根据游戏世界是基于关卡的还是随机 3D 的，将其划分成方块或立方体。然后，随机地在游戏世界中选择数千、数万或数十万个视点，或者根据玩家的移动路线均匀地选择视点；再从每个视点发出到游戏世界中每个多边形的射线。如果至少有一条从视点出发的射线能够不受阻碍地到达多边形的一个顶点，则可以确定从视点看，该多边形是可见的，进而将其加入到视点的 PVS 中。图 13.54 说明了这一点。

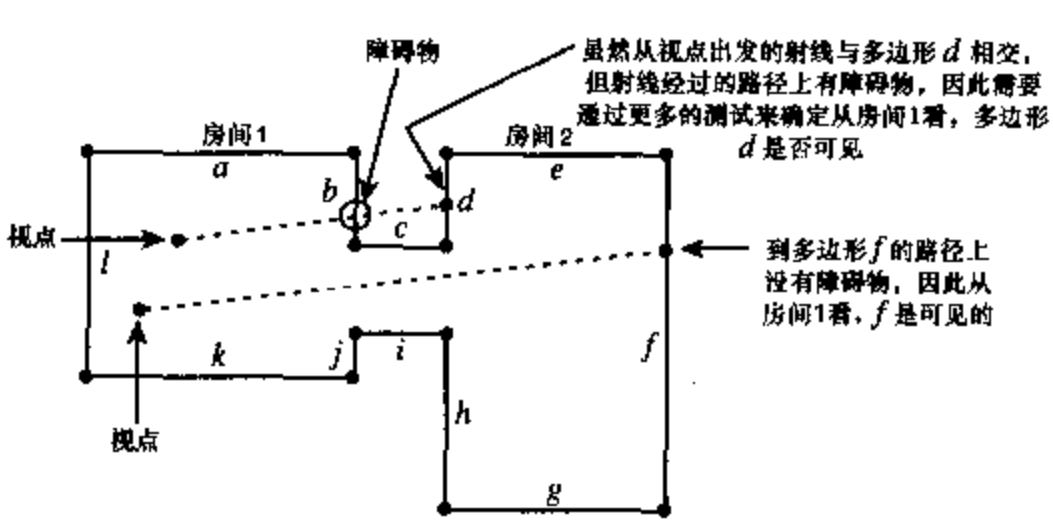


图 13.54　使用光线投射来确定可见性

3. 有关 PVS 的最后思考

对于使用大型多边形数据库的游戏而言，潜在可见集是必不可少的。在运行阶段使用它们的开销几乎为零，同时即使使用最粗糙的编码方法，它们占用的内存量充其量是中型的。PVS 与 BSP 树、八叉树、（尤其是）入口等协同工作。PVS 的计算量可能非常大，这取决于您对 PVS 的精度要求。然而，正如前面指出的，简单的基于工具的 PVS 或基于入口/光线投射的 PVS 只需进行少量的几何和光线跟踪计算。最后，计算出 PVS 后，便可以使用邻接矩阵、数组或位表对其进行编码。

13.5　入　　口

在过去的几年中，入口（portal）吸引了大量的眼球。有趣的是，其首次出现可追溯到 60 年代，因此很多编码员都认为它是自己发明的。两个最著名的入口游戏是 Descent 和 Duke Nukem 3D，如图 13.55 和 13.56 所示。Descent 是一款 3D 飞行游戏，支持 6 种飞行方向，让玩家能够在好像一望无际的 3D 隧道迷宫

中飞行。它支持光照、纹理映射具有非常棒的效果，但令人惊讶的是，游戏世界的渲染速度却非常快。Duke Nukem 3D 是 90 年代中期推出的另一款非常精彩的 3D 游戏，它支持实时光照、纹理映射、多关卡、晋级（stairs）、移动物体等特性，也是一款使用入口技术的产品。入口技术并不神奇，它只不过是一种实时实现的 PVS 策略。

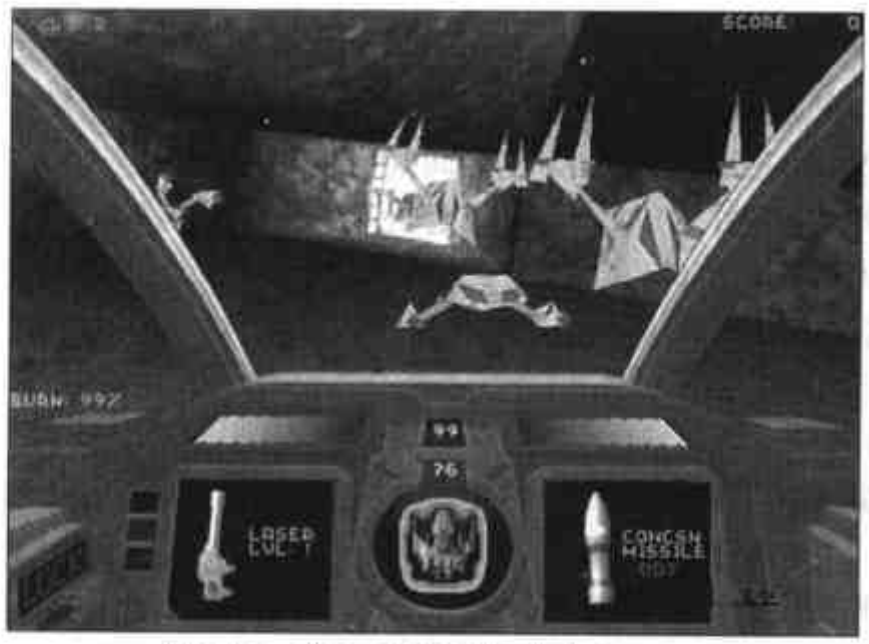


图 13.55　将入口使用到极致的 Descent

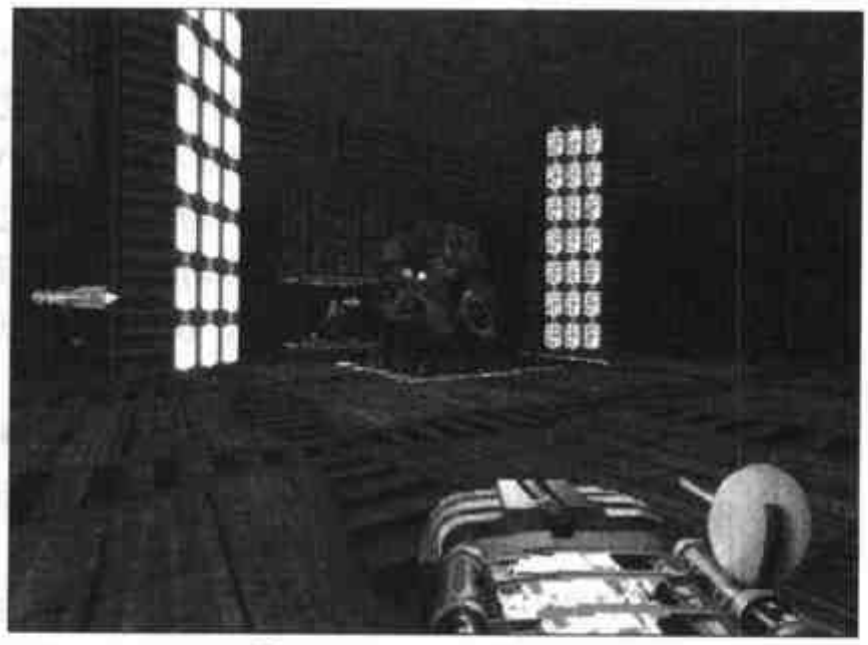


图 13.56　Duke Nukem 3D

关键之处在于，即使关卡中有 10 亿个多边形，如果每帧只需绘制两三百个多边形，游戏将非常精彩，好像能够处理大量的数据。其中的技巧在于，只渲染能够看到的那些多边形。入口技术基于这样一个事实：如果环境是基于房间或单元格的，并能够标记到邻接房间的“入口”，可以使用一种巧妙的技术来动态地计算 PVS。下面介绍其工作原理。请看图 13.57，其中有 5 个房间，每扇门都被标记为入口。

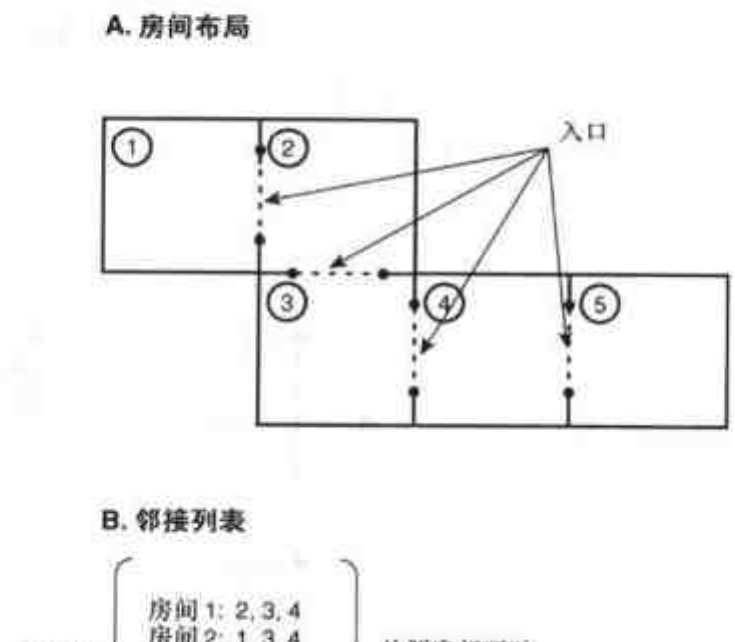


图 13.57　根据入口离线计算房间到房间的可见性

对于房间中的任何物体，来看看能否根据入口确定哪些多边形是可见的。首先，至少离线地创建一个房间到房间的可见性列表。之所以这样做是因为，如果从房间 1 看不到房间 5，则没有必要对房间 5 应用入口逻辑。这里使用邻接列表，而不是邻接矩阵，如图 13.57b 所示。有了这些列表后，便可以实时地进行渲染了，步骤如下：

1．确定视点所在的房间/单元格，并保存这种信息。

2．使用任何技术（BSP、z 缓存、z 排序等）渲染当前房间中的多边形。

3．对于当前房间的每个入口（我们有一个这样的列表），将定义该入口的平面作为附加裁剪面，来调整视景体，然后进入入口连接的房间，并绘制该房间中的多边形。

深入探讨裁剪细节之前，先来详细介绍上述处理过程。从视点所在的房间开始，渲染该房间中的多边形，然后通过入口列表递归地遍历该房间的邻接列表，但每次进入下一个房间时，都只渲染该房间中从源房间通过入口可以看到的多边形。换句话说，根据入口对视景体进行裁剪，如图 13.58 所示。

下面通过一个 2D 空间中包含两个房间的例子来说明这个处理过程。如图 13.58 所示，房间 1 的邻接列表中包含房间 2，这意味着通过入口这两个房间之间是相互可见的。视点位于房间 1 中，因此根据原始视景体渲染房间 1 中所有的多边形。接下来进入房间 2，因为通过入口 A 可以看到它。然而，需要根据入口的几何形状（这里是 2D 的）对视景体进行调整，新的视景体如图 13.58b 所示。从视点看，只有位于新视景体内的多边形是可见的，其他多边形不可见，不会被渲染。由于根据入口的形状对视景体进行了裁剪，根据定义，从房间 1 中的视点只能看到房间 2 中通过入口可见的多边形。

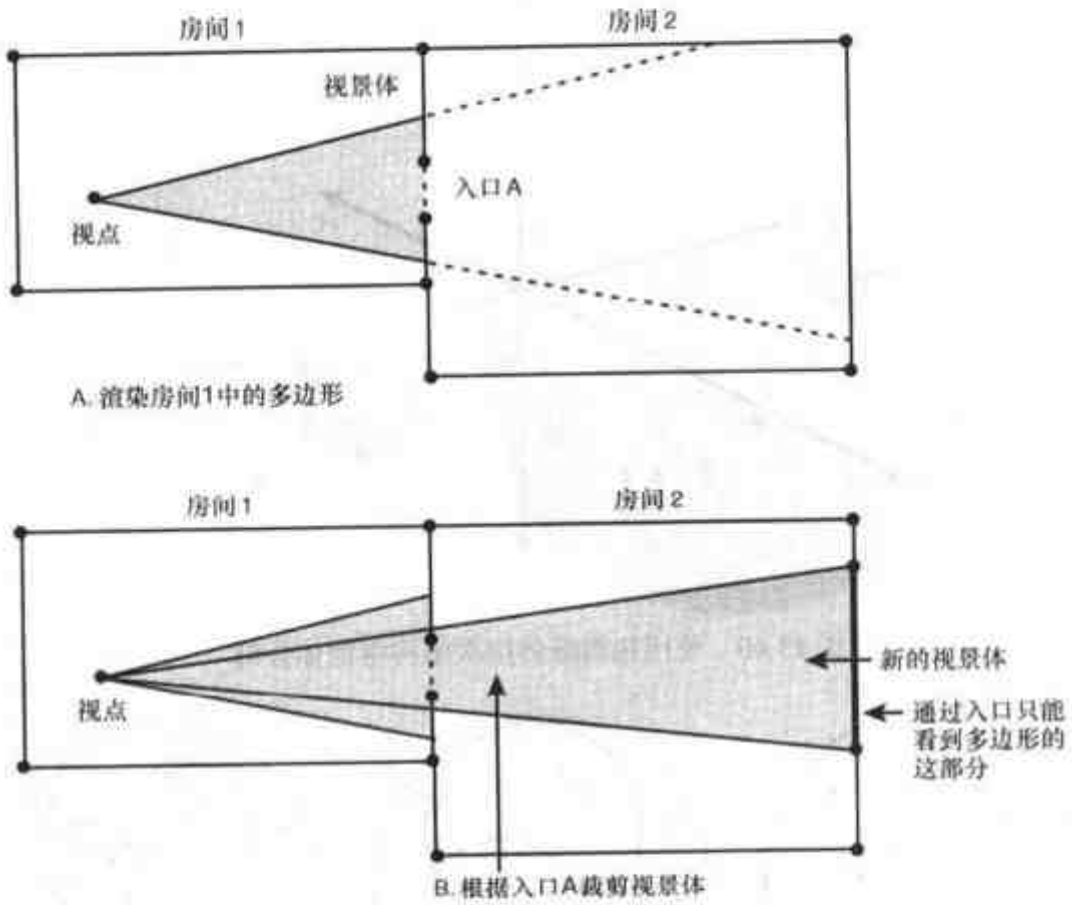


图 13.58　包含两个房间的入口裁剪和遍历范例

刚才介绍的入口技术存在两个问题。首先，没有提供 PVS，而只是创建了一个新的视景体。然而，如果有 BSP 树或八叉树，可以根据当前入口剔除整个房间或单元格，从而节省时间。第二个问题更复杂。每次递归到下一个房间，根据入口的几何形状调整视景体后，视景体将从 4 面（还有近裁剪面和远裁剪面）变成 n 面，如图 13.59 所示。

这没什么，可以使用一组平面来实现；真正的问题在于调整视景体后的裁剪：裁剪时间的增加开始抵消渲染时间的减少——裁剪时间如此之长，还不如直接渲染所有的多边形呢！

总之，对于通过过道相连或形状非常规则的游戏世界，入口技术的效果很不错。另外，如果愿意放松裁剪和使用入口平面调整视景体的条件，将可以得到非常高的实时性能。然而，入口技术还是不如 PVS——它怎么可能与 PVS 相比呢？预先知道什么需要绘制什么不需要绘制总是更好。

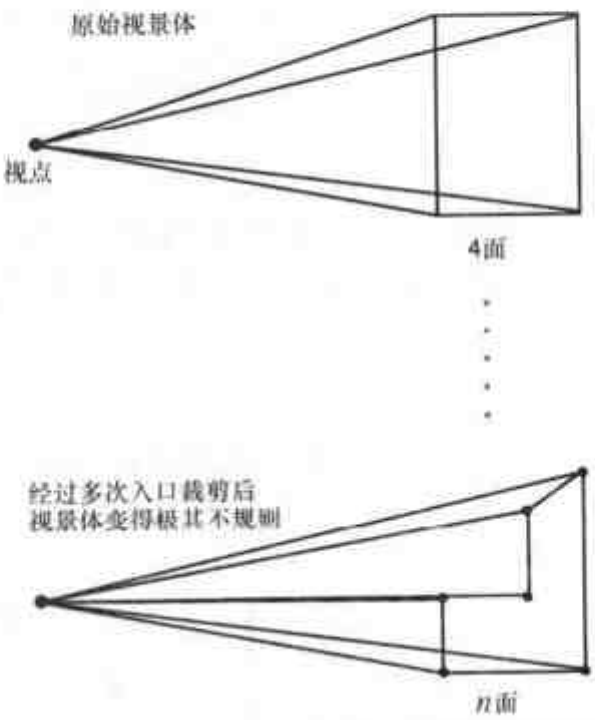


图 13.59　经过多次入口裁剪后视景体变得极其不规则

13.6　包围体层次结构和八叉树

包围体层次结构（bounding hierarchical volumes，BHV）只不过是另一种对物体或几何体进行分组，以实现快速碰撞检测或剔除的方法。事实上，我们从开始到现在一直在使用包围球来进行物体的快速剔除。BHV 只是深化了这种思想，创建一个包围体层次结构，如图 13.60 所示。包围体可以是如图 13.60 所示的球体，可以是平行于坐标轴的立方体，也可以是基于物体的立方体，选择权在您手中。在大多数情况下，使用球体或与坐标轴平行的立方体。

例如，假设有一组如图 13.61 所示的物体，其中每个物体用其包围球表示。BPV 的理念是，将物体组合成大型集合，这样便可以对整个集合而不是各个组成部分进行剔除。在图 13.61b 中，物体被组合到较小的球体中，而在图 13.61c 中，物体被组合到更小的球体中。

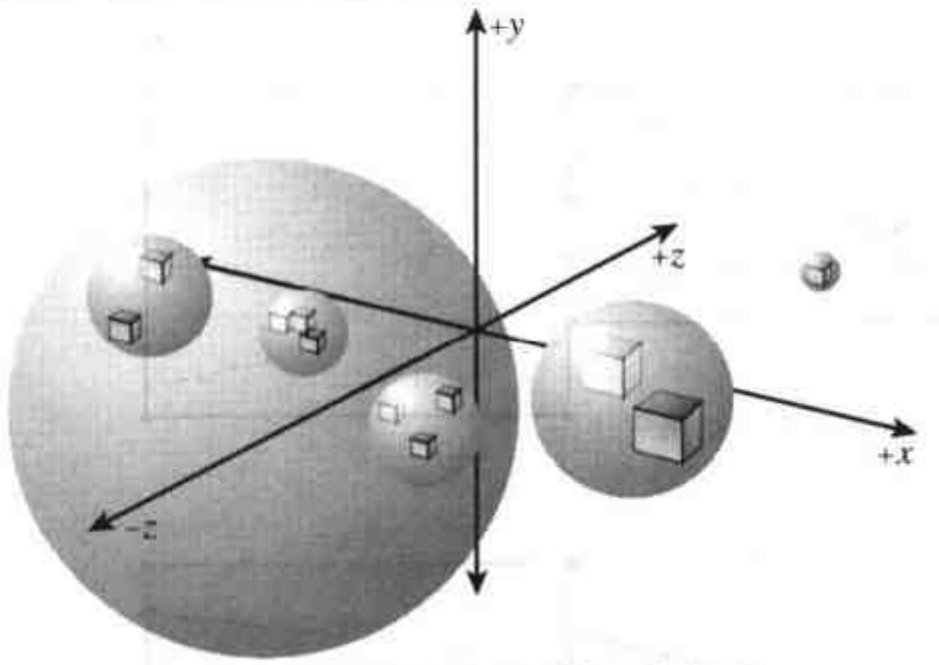


图 13.60　使用包围球体层次结构将物体分组

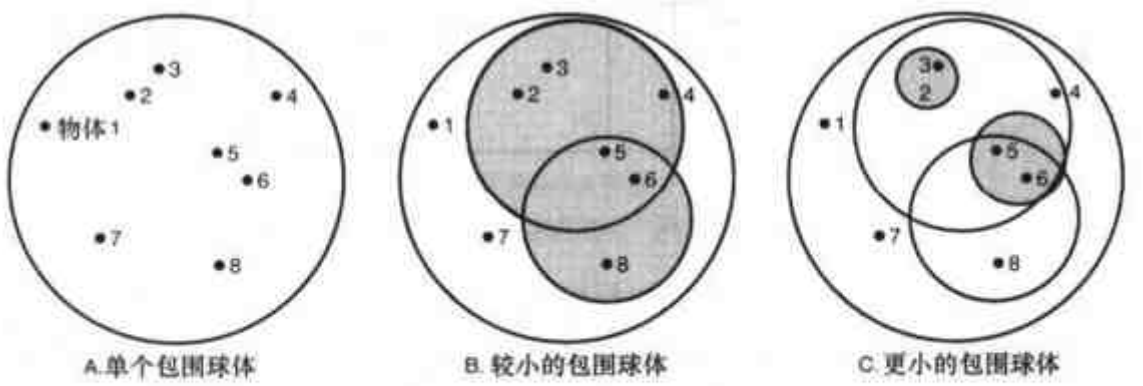


图 13.61　包围体层次结构

可以用类似于图 13.62 的任何数据结构来表示 BHV，但树形结构最合适。然而，在层次结构中，节点的子节点可能超过两个，因此需要使用 *n*-叉树结构，如图 13.62 所示。

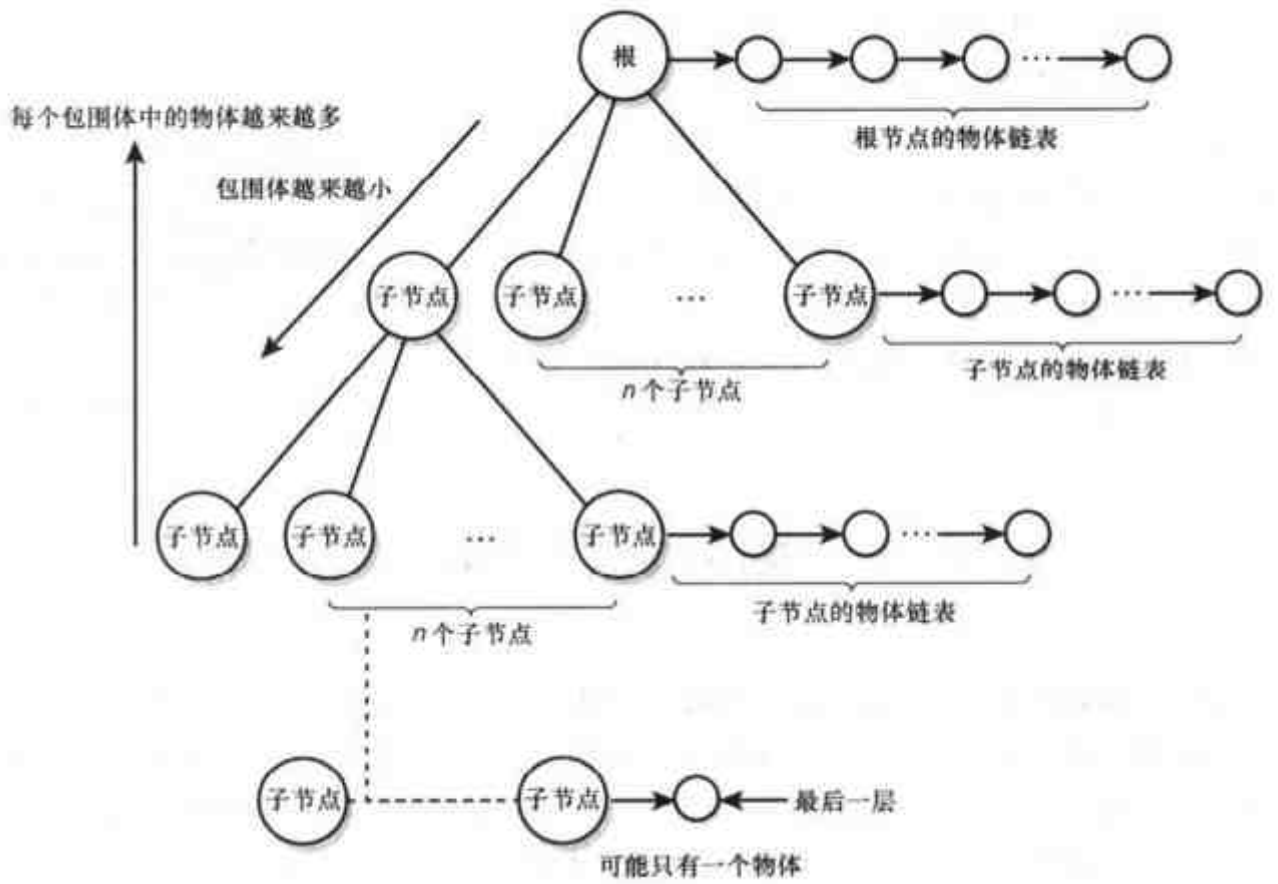


图 13.62　一种用于表示 BHV 的数据结构

每个节点都有球心、半径、指向子节点的指针数组、节点中的物体链表（数组）以及状态信息。下面是可能使用的一套数据结构：

```
// 用于存储物体的物体容器
typedef struct OBJ_CONTAINERV1_TYP
```

```
    {
    int state;     // 状态
    int attr;      // 属性
    POINT4D pos;    // 位置
    VECTOR4D vel;    // 速度
    VECTOR4D rot;    // 旋转速度
    int auxi[8];    // 包含 8 个元素的 int 辅助数组
    int auxf[8];    // 包含 8 元素的 float 辅助数组
    void *aux_ptr;   // 辅助指针
    void *obj;     // 指向 OBJECT4DV2 的指针

    } OBJ_CONTAINERV1, *OBJ_CONTAINERV1_PTR;

// BHV 节点数据结构
typedef struct BHV_NODEV1_TYP
    {
    int state;     // 节点的状态
    int attr;     // 节点的属性
    POINT4D pos;    // 中心
    VECTOR4D radius;  // 沿 x、y、z 轴的半径
    int num_objects;  // 节点中包含多少个物体
    OBJ_CONTAINERV1 *objects[MAX_OBJECTS_PER_BHV_NODE]; // 物体数组

    int num_children; // 子节点数

    BHV_NODEV1_TYP *links[MAX_BHV_PER_NODE]; // 指向子节点的指针数组

    } BHV_NODEV1, *BHV_NODEV1_PTR;
```

上述数据结构适用于包围体为球体、立方体或长方体的情形。半径是用 VECTOR4D 表示的，包围体沿每个轴的长度可以不同。字段 radius 和 attr 一起，使得 BHV_NODEV1 适用于各种不同包围体的情形。将 BHV_NODEV1 做了细微的抽象化，它包含一个更通用的 OBJ_CONTAINERV1，而不是 OBJECT4DV2。另外，OBJ_CONTAINERV1 有一个 void 指针，用于指向一个 OBJECT4DV2，这样，即使 OBJECT4DV2 被修改，仍可以使用上述数据结构。

当然，我们需要一种创建 BHV 树的策略，即在每层如何选择包围球的位置和大小。不过，在此之前，先来讨论如何使用 BHV 树。

13.6.1　使用 BHV 树

假设已经使用包围球（或立方体）创建好了 BHV 树，我们便可以在运行阶段使用它来进行大规模的碰撞检测或剔除。它们之间是彼此相关的，本章的主题是渲染而不是碰撞检测，因此这里只讨论如何使用 BHV 树来进行剔除。

BHV 树的每一层以不同的粒度（coarseness）表示了游戏世界中所有的几何体。在最上面一层，只有一个球体，它包围了整个游戏世界，因此第一步是根据视景体对整个游戏世界进行剔除。这很容易，类似于基于包围球来剔除物体。如果整个根节点都被剔除，则将数组 objects[ ]中所有的物体都标记为被剔除，无需再进行其他处理。给所有物体设置剔除标记的代码如下：

```
SET_BIT(obj->state, OBJECT4DV2_STATE_CULLED);
```

如果最上面的节点不能被剔除，则遍历子节点，对每个子节点的包围球进行剔除。在每个节点处，可以以深度优先或广度优先的方式递归进行上述处理，如图 13.63 所示。无论采用哪种方式，必要时都能检测整个 BHV 树。读者需要记住的是，如果节点被剔除，该节点中所有的物体都被标记为被剔除，不再遍历其任何子节点。

处理完毕后，每个被剔除的物体都被相应地标记，这样渲染时，在遍历物体的主循环中，可以通过检查 OBJECT4DV2_STATE_CULLED 标记来确定物体是否被剔除。如果物体未被剔除，仍必须执行标准的物体剔除处理，因为 BHV 系统不一定会创建只包含一个物体的包围体。

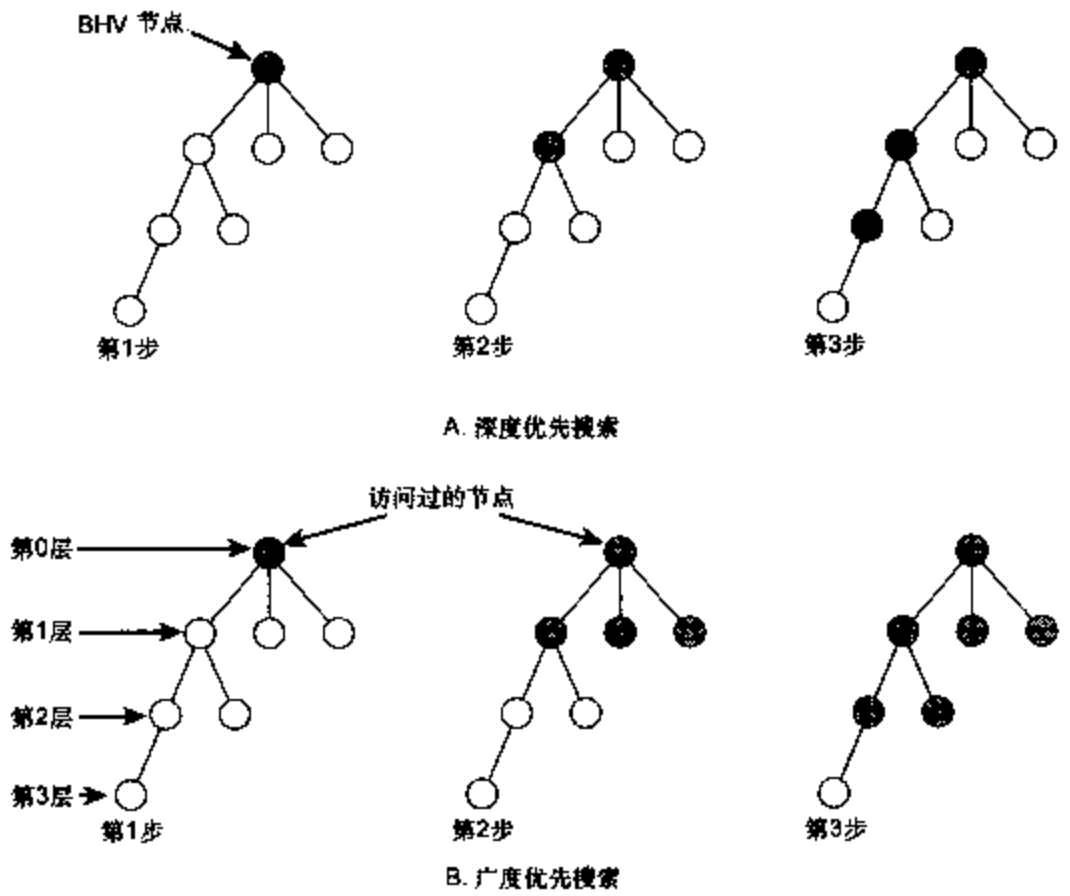


图 13.63　BHV 树的深度优先和广度优先搜索

13.6.2　运行性能

接下来讨论运行性能以及 BHV 为何有助于提高性能。假设有 10 000 个物体，它们均匀地分布在空间中。如果有 4 层包围体，第 0 层为 BHV 树根，包含所有的物体；第 1 层也包含所有的物体，但每个包围体的半径最大为根包围体的 50%，依次类推。

具体情况如下：

- **根**——包围体的半径为 r，包含全部物体。
- **第 1 层**——包围体的半径<=r/2。这一层有一个或多个包围体，我们假设有 5 个，半径都是 r/2，它们在空间中均匀分布，因此每个包围体中有 2000 个物体。
- **第 2 层**——包围体的半径<=r/4。这一层有一个或多个包围体，我们假设有 20 个，半径都是 r/4，它们在空间中均匀分布，因此每个包围体中有 100 个物体。
- **第 3 层**——包围体的半径<=r/8。这一层有一个或多个包围体，我们假设有 200 个，半径都是 r/8，它们在空间中均匀分布，因此每个包围体中有 50 个物体。

这样，BHV 树如图 13.64 所示。接下来将该 BHV 树传递给根据视景体对包围球进行剔除的函数（其代码与 Cull_OBJECT4DV2() 相同）。如果能够根据视景体剔除树根，便可以知道没有一个物体是可见的——原本需要进行 10 000 次包围球测试，现在只需一次测试！然而，除非太阳从西边出来（hell has frozen over），否则不可能整个游戏世界都不可见。我们必须检测 BHV 树的下一层，这是有趣的地方。

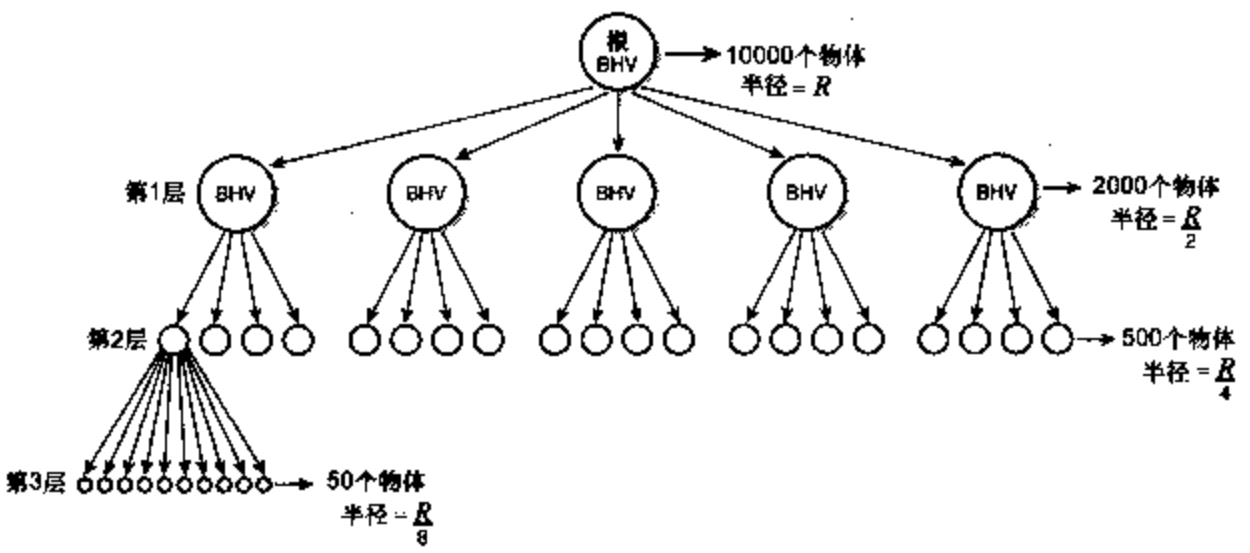


图 13.64　BHV 树范例

根有 5 个子节点，因此需要以深度优先或广度优先的方式遍历这 5 个子节点。我们暂时使用广度优先的方式，这意味着需要依次判断 5 个子节点是否被剔除。如果是，则将节点中所有的物体都标记为被剔除，并将节点标记为访问过，这样便不会访问其子节点。假设 5 个节点中有 4 个被剔除。接下来进入下一层，这需要处理 4 个节点。这个过程将不断重复下去，直到每个需要访问的节点都访问过为止。

遍历结束后，所有可能被剔除的物体都已剔除。一般而言，需要执行几十次剔除测试，相对于原本需要进行 10 000 次测试，这是非常有意义的。这就是 BHV 技术的威力所在。

然而，显然还有一些可优化的地方。例如，遍历 BHV 树时，同一个物体很可能包含在多个节点中，可能在高层被剔除后再被剔除（或试图将其剔除）。在大多数情况下，这是可以接受的，这主要与任何两个节点之间的重复率有关。

第二个重要的优化方法是，使用一个计数器来记录有多少个物体被剔除（重复剔除只算一次），当计数器等于物体数目后，结束遍历。

13.6.3　选择策略

最后一个难题是选择包围体。本章后面的一节将讨论八叉树，其确定性非常高。广义 BHV 本质上是确定性或试探性的。例如，可以使用工具来手工选择包围体、编写不那么精确的试探算法或编写一种确定性非常高的算法。这里只介绍一些选择包围体的思路。

另外，很多人使用 BHV 来对移动物体进行实时地碰撞检测和剔除，这意味着每当物体离开节点时都必须重新生成 BHV 树，或者每帧都重新生成 BHV 树，采取哪种方法取决于哪种方法更合适。因此，通常使用只有 3～8 层的简单方案。别忘了，我们只是使用 BHV 来提供一些帮助，并不要求它是完美的；而速度始终是最重要的考虑因素。

1. 分而治之

这种策略以均匀的方式将 3D 空间划分为立方体、球体等，每个分区的大小相同。例如，第一步总是将整个空间包含在根 BHV 节点中；第二步将空间均匀地划分为大小相同的立方体——可能是 16×16×16 或 4×4×4 的，然后据此确定包围体的大小。不包含任何物体的分区将不会作为 BHV 节点加入到这一层中。接下来，再次将每个 BHV 节点划分成立方体（或其他几何形状），如图 13.65a 和 13.65b 所示。这种分而治之技术的优点是易于实现，这与八叉树一脉相承（实际上，这样做将创建八叉树）。

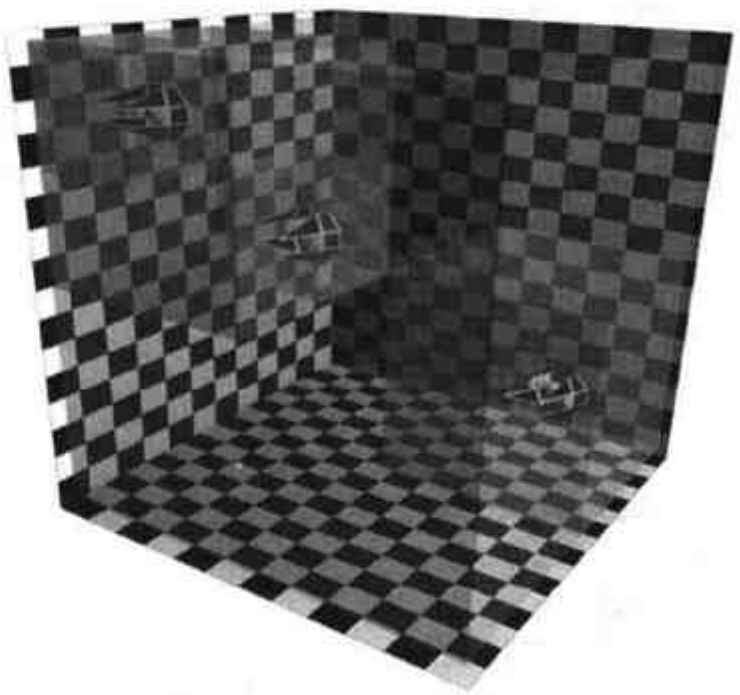


图 13.65a　3D 分区的实心视图

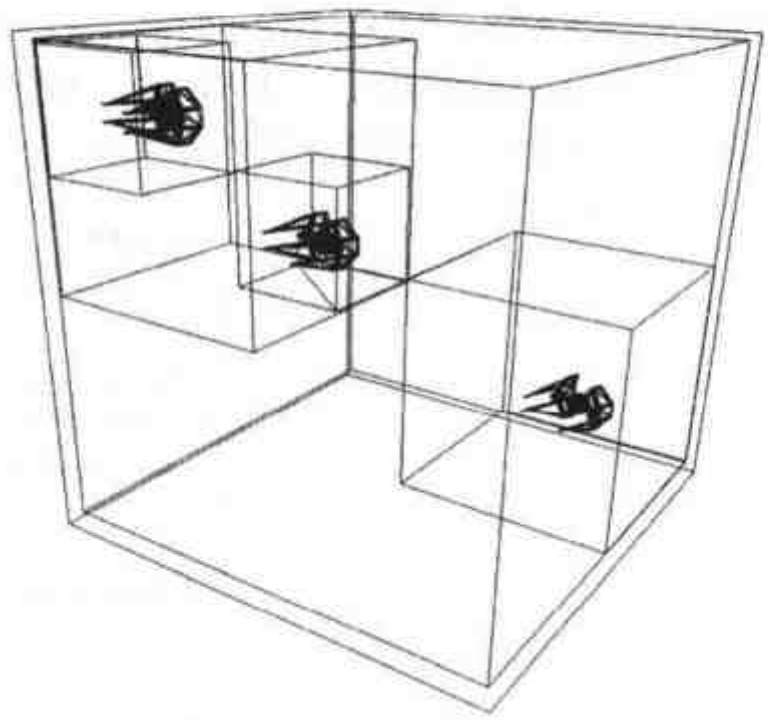


图 13.65b　3D 分区的线框视图

2. 集群技术

虽然分而治之技术易于实现，但不太灵巧。例如，对于图 13.66 所示的游戏世界，对其进行分区无疑是浪费时间，因为很容易将其划分成三个物体群集。

集群技术使用一种试探算法，它随机地选择一个物体，将离该物体的距离小于集群半径的所有物体作为一个潜在集群，并计算集群中其他物体到该物体的平均半径。然后，继续随机地选择其他物体作为集群种子（cluster seed），并执行同样的逻辑。

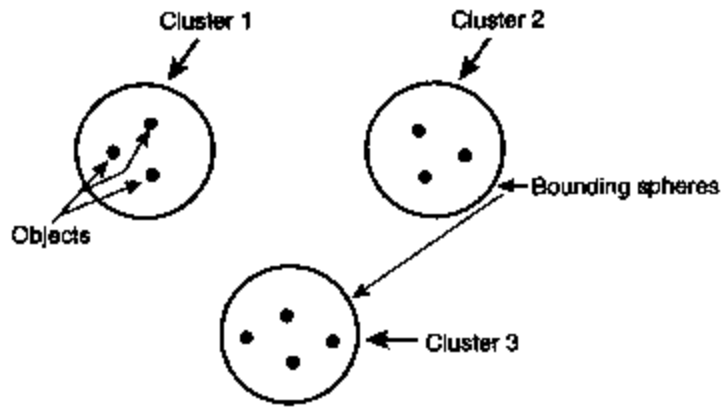


图 13.66　没必要进行分区的情形

这种过程不断重复下去，直到满足终止条件（可能是选择了 n 个集群种子等）为止。接下来，将最佳的潜在集群作为 BHV 节点，然后以类似的方法找出其他的 BHV 节点。图 13.67 以高度简化的方式说明了这种处理方法，其中有 6 个物体。首先，将每个物体都作为集群种子，发现物体 1 是最佳的，因为它到集群中其他物体的平均距离最小。因此，以物体 1 为中心创建第一个 BHV 节点。接下来，从余下的物体选择一个最佳的，将其作为中心来创建下一个 BHV 节点，依次类推。

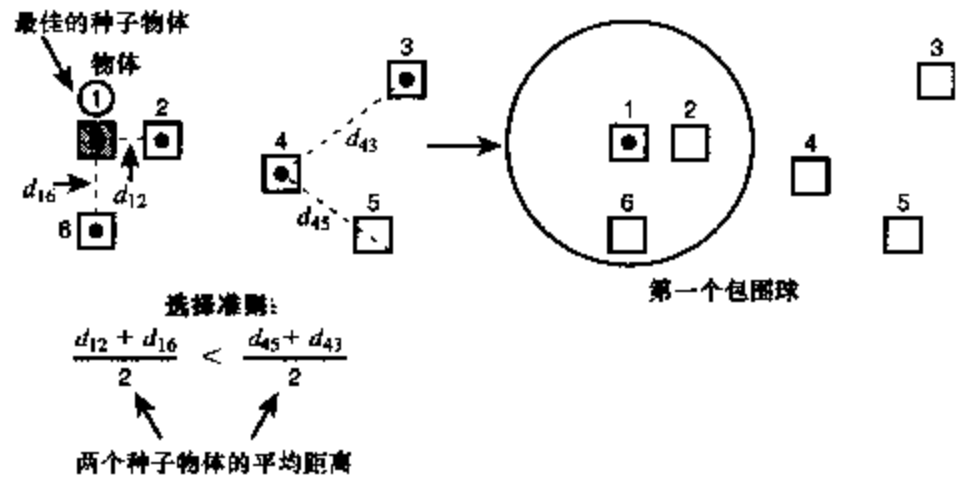


图 13.67　根据到集群中其他物体的平均距离最小来选择种子物体

13.6.4　实现 BHV

作者编写了一个演示 BHV 的程序，它创建一个 3D 世界，加载几百个物体，并将它们随机地放置在 3D 世界中。然后，使用分而治之算法将这些物体分组以创建 BHV 树，BHV 树包含 n 层，其中 n 是可以通过函数参数指定的。执行 BHV 分析的函数是专门针对物体的表示方式而设计的。如果读者查看多物体演示程序的代码将发现，大多数情况下，都只加载了一个物体，然后定义了一个用于物体位置的数组，其中每个元素都指向加载的物体。在 BHV 演示程序中，也使用了这种技巧：创建一个虚拟物体数组，然后根据其中每个元素指定的位置渲染同一个物体。图 13.68 说明了这个虚拟物体数组。

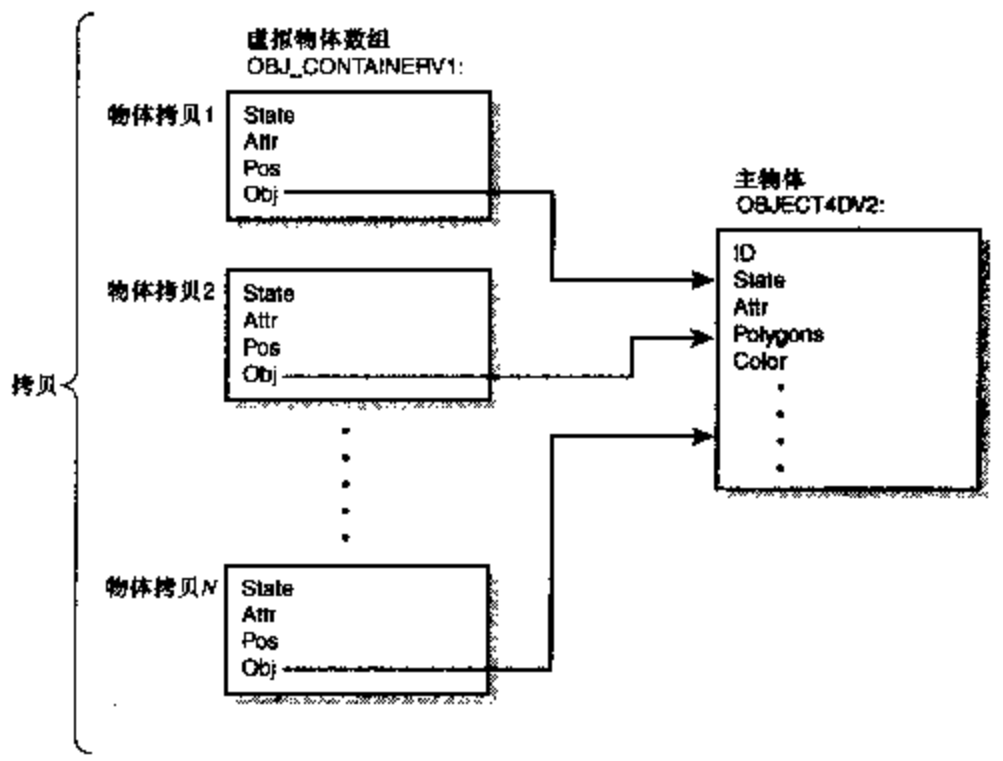


图 13.68　同一个物体的多个拷贝

下面是物体容器的定义：

```
typedef struct OBJ_CONTAINERV1_TYP
{
int state;     // 状态
int attr;      // 属性
POINT4D pos;    // 位置
OBJECT4DV2_PTR obj; // 指向 OBJECT4DV2 的指针

} OBJ_CONTAINERV1, * OBJ_CONTAINERV1_PTR;
```

当然，前面定义的 BHV_NODEV1 被用来表示 BHV 节点。BHV 处理涉及的函数只有两个：一个用于创建 BHV 树，另一个用于根据视景体对 BHV 树进行剔除。创建 BHV 树的函数如下：

```
void BHV_Build_Tree(BHV_NODEV1_PTR bhv_tree,        // 要创建的 BHV 树
   OBJ_CONTAINERV1_PTR bhv_objects, // 指向场景中物体数组的指针
   int num_objects,          // 场景中的物体数目
   int level,             // BHV 树应包含多少层
   int num_divisions,        // 沿每个轴将节点分割成多少份
   int universe_radius)        // 游戏世界的大小
{
// 这个函数使用分而治之算法将物体分组，以创建 BHV 树

Write_Error("\nEntering BHV function...");

// 创建树根吗?
if (level == 0)
  {
  Write_Error("\nlevel = 0");

  // 将节点位置设置为(0, 0, 0)
  bhv_tree->pos.x = 0;
  bhv_tree->pos.y = 0;
  bhv_tree->pos.z = 0;
  bhv_tree->pos.w = 1;

// 将节点半径设置为游戏世界的半径
bhv_tree->radius.x = universe_radius;
bhv_tree->radius.y = universe_radius;
bhv_tree->radius.z = universe_radius;
bhv_tree->radius.w = 1;

Write_Error("\nnode pos[%f, %f, %f], r[%f, %f, %f]",
         bhv_tree->pos.x,bhv_tree->pos.y,bhv_tree->pos.z,
         bhv_tree->radius.x,bhv_tree->radius.y,bhv_tree->radius.z);

// 创建树根节点，将所有物体都加入到该节点中
for (int index = 0; index < num_objects; index++)
 {
 // 判断物体是否已经被剔除
 if (!(bhv_objects[index].state & OBJECT4DV2_STATE_CULLED))
  {
  bhv_tree->objects[bhv_tree->num_objects++] =
       (OBJ_CONTAINERV1_PTR)&bhv_objects[index];
  } // end if

 } // end for index

// 将所有物体加入到根节点中后，将该节点设置为活动状态
bhv_tree->state = 1;
bhv_tree->attr  = 0;

// 将所有指向了节点的指针都设置为 NULL
for (int ilink = 0; ilink < MAX_BHV_PER_NODE; ilink++)
  bhv_tree->links[ilink] = NULL;

// 设置包含的物体数目
```

```
    bhv_tree->num_objects = num_objects;

    Write_Error("\nInserted %d objects into root node", bhv_tree->num_objects);
    Write_Error("\nMaking recursive call with root node...");

    // 递归地创建BHV树的其他部分
    BHV_Build_Tree(bhv_tree,
           bhv_objects,
           num_objects,
           1,
           num_divisions,
           universe_radius);
   } // end if
 else
   {
   Write_Error("\nEntering Level = %d > 0 block, number of objects = %d",
        level, bhv_tree->num_objects);

   // 检测结束状态
   if (bhv_tree->num_objects <= MIN_OBJECTS_PER_BHV_CELL)
    return;

   // 创建子节点
   // 必须将当前节点划分成多个子节点
   // 然后将物体插入到每个子节点中，再递归调用函数，以创建BHV子树

 // 创建临时3D单元格
 BHV_CELL cells[MAX_BHV_CELL_DIVISIONS]
        [MAX_BHV_CELL_DIVISIONS]
        [MAX_BHV_CELL_DIVISIONS];

 // 根据半径和中心找出包围体的原点
 int x0 = bhv_tree->pos.x - bhv_tree->radius.x;
 int y0 = bhv_tree->pos.y - bhv_tree->radius.y;
 int z0 = bhv_tree->pos.z - bhv_tree->radius.z;

 // 计算单元格沿x、y、z轴的大小
 float cell_size_x = 2*bhv_tree->radius.x / (float)num_divisions;
 float cell_size_y = 2*bhv_tree->radius.y / (float)num_divisions;
 float cell_size_z = 2*bhv_tree->radius.z / (float)num_divisions;

 Write_Error("\ncell pos=(%d, %d, %d) size=(%f, %f, %f)",
       x0,y0,z0, cell_size_x, cell_size_y, cell_size_z);

 int cell_x, cell_y, cell_z; // 用于存储单元格在3D阵列中的位置

 // 清空单元格内存
 memset(cells, 0, sizeof(cells));

 //沿每条轴将空间分成num_divisions (必须小于MAX_BHV_CELL_DIVISIONS)等份
 // 确定物体的中心位于哪个单元格中
 for (int obj_index = 0; obj_index < bhv_tree->num_objects; obj_index++)
   {
   // 计算物体位于哪个单元格中
   cell_x = (bhv_tree->objects[obj_index]->pos.x - x0)/cell_size_x;
   cell_y = (bhv_tree->objects[obj_index]->pos.y - y0)/cell_size_y;
   cell_z = (bhv_tree->objects[obj_index]->pos.z - z0)/cell_size_z;

   // 将物体插入到单元格的物体数组中
  cells[cell_x][cell_y][cell_z].obj_list[cells[cell_x]
   [cell_y][cell_z].num_objects] = bhv_tree->objects[obj_index];

   Write_Error("\ninserting object %d
   located at (%f, %f, %f) into cell (%d, %d, %d)",
        obj_index,
        bhv_tree->objects[obj_index]->pos.x,
        bhv_tree->objects[obj_index]->pos.y,
```

```
        bhv_tree->objects[obj_index]->pos.z,
        cell_x, cell_y, cell_z);

    // 将单元格中的物体数加 1
    if (++cells[cell_x][cell_y][cell_z].num_objects >=
      MAX_OBJECTS_PER_BHV_CELL)
  cells[cell_x][cell_y][cell_z].num_objects  = MAX_OBJECTS_PER_BHV_CELL-1;

    } // end for obj_index

  Write_Error("\nEntering sorting section...");

// 有了单元格后，为每个不为空的单元格创建一个 BHV 节点

for (int icell_x = 0; icell_x < num_divisions; icell_x++)
  {
  for (int icell_y = 0; icell_y < num_divisions; icell_y++)
    {
    for (int icell_z = 0; icell_z < num_divisions; icell_z++)
      {
      // 单元格中有物体吗?
      if ( cells[icell_x][icell_y][icell_z].num_objects > 0)
        {
        Write_Error("\nCell %d, %d, %d contains %d objects",
              icell_x, icell_y, icell_z,
              cells[icell_x][icell_y][icell_z].num_objects);

        Write_Error("\nCreating child node...");

        // 创建一个节点，并将指针指向它
        bhv_tree->links[ bhv_tree->num_children ] =
         (BHV_NODEV1_PTR)malloc(sizeof(BHV_NODEV1));

        // 清空节点的内容
memset(bhv_tree->links[ bhv_tree->num_children ],
   0, sizeof(BHV_NODEV1) );

// 设置节点
BHV_NODEV1_PTR curr_node =
  bhv_tree->links[ bhv_tree->num_children ];

// 设置位置
curr_node->pos.x = (icell_x*cell_size_x + cell_size_x/2) + x0;
curr_node->pos.y = (icell_y*cell_size_y + cell_size_y/2) + y0;
curr_node->pos.z = (icell_z*cell_size_z + cell_size_z/2) + z0;
curr_node->pos.w = 1;

//半径为 cell_size / 2
curr_node->radius.x = cell_size_x/2;
curr_node->radius.y = cell_size_y/2;
curr_node->radius.z = cell_size_z/2;
curr_node->radius.w = 1;

// 设置物体数
curr_node->num_objects =
  cells[icell_x][icell_y][icell_z].num_objects;

// set num children
curr_node->num_children = 0;

// 设置状态和属性
curr_node->state     = 1; // 将节点设置为活动状态
curr_node->attr      = 0;

// 将物体插入到节点的物体列表中
for (int icell_index = 0;
  icell_index < curr_node->num_objects; icell_index++)
```

```
            {
            curr_node->objects[icell_index] =
            cells[icell_x][icell_y][icell_z].obj_list[icell_index];
            } // end for icell_index

          Write_Error("\nChild node pos=(%f, %f, %f), r=(%f, %f, %f)",
           curr_node->pos.x,curr_node->pos.y,curr_node->pos.z,
           curr_node->radius.x,curr_node->radius.y,curr_node->radius.z);

          // 将子节点数加 1
          bhv_tree->num_children++;

          } // end if

        } // end for icell_z

      } // end for icell_y

    } // end for icell_x

    Write_Error("\nParent has %d children..", bhv_tree->num_children);

    // 为每个节点创建一个 BHV 子树
    for (int inode = 0; inode < bhv_tree->num_children; inode++)
      {
      Write_Error("\nfor Level %d, creating child %d", level, inode);

      BHV_Build_Tree(bhv_tree->links[inode],
           NULL, // unused now
           NULL, // unused now
           level+1,
           num_divisions,
           universe_radius);

       } // end if

         } // end if

      } // end else level > 0

    Write_Error("\nExiting BHV...level = %d", level);

    } // end BHV_Build_Tree
```

下面是根据视景体对 BHV 树进行剔除的函数：

```
int BHV_FrustrumCull(BHV_NODEV1_PTR bhv_tree, // BHV 树根
     CAM4DV1_PTR cam,      // 相机
     int cull_flags)      // 要考虑的裁剪面
{
// 这个函数是基于矩阵的
// 它根据传入的相机信息对 BHV 树执行视景体剔除
// cull_flags 参数指定需要根据哪些裁剪面进行剔除
// 在对 BHV 进行剔除时，将修改节点的状态信息，供渲染函数参考

// 检查 BHV 树和相机是否有效
if (!bhv_tree || !cam)
  return(0);

// 需要遍历 BHV 树，并进行剔除

// 第 1 步：将节点的包围球球心变换为相机坐标

POINT4D sphere_pos; // 用于存储包围球球心的变换结果

// 对球心坐标进行变换
Mat_Mul_VECTOR4D_4X4(&bhv_tree->pos, &cam->mcam, &sphere_pos);

// 第 2 步：根据剔除标记对节点进行剔除
if (cull_flags & CULL_OBJECT_Z_PLANE)
```

```
{
// cull only based on z clipping planes

// 远裁剪面测试
if ( ((sphere_pos.z - bhv_tree->radius.z) > cam->far_clip_z) ||
   ((sphere_pos.z + bhv_tree->radius.z) < cam->near_clip_z) )
  {
  // 整个节点被剔除，需要设置其中每个物体的剔除标记
  for (int iobject = 0; iobject < bhv_tree->num_objects; iobject++)
    {
    SET_BIT(bhv_tree->objects[iobject]->state, OBJECT4DV2_STATE_CULLED);
    } // end for iobject

  // 该节点访问过且被剔除
  bhv_nodes_visited++;

  return(1);
  } // end if

} // end if

if (cull_flags & CULL_OBJECT_X_PLANE)
{
// 只根据左、右裁剪面进行剔除
// 本可以使用平面方程，但使用三角形相似更容易，因为这是一个 2D 问题
// 如果视角为 90 度，问题将更简单，但这里假设不是

// 根据右裁剪面和左裁剪面来检测包围球上最右边和最左边的点
float z_test = (0.5)*cam->viewplane_width*sphere_pos.z/cam->view_dist;

if ( ((sphere_pos.x - bhv_tree->radius.x) > z_test)  || //位于视景体右边
   ((sphere_pos.x + bhv_tree->radius.x) < -z_test) )  // 位于视景体左边
  {
  // 整个节点被剔除，需要设置节点中每个物体的剔除标记
  for (int iobject = 0; iobject < bhv_tree->num_objects; iobject++)
    {
    SET_BIT(bhv_tree->objects[iobject]->state, OBJECT4DV2_STATE_CULLED);
    } // end for iobject

  // 这个节点被访问过且被剔除
  bhv_nodes_visited++;

  return(1);
  } // end if
} // end if

if (cull_flags & CULL_OBJECT_Y_PLANE)
{
// 只根据上、下裁剪面来进行剔除
// 本可以使用平面方程，但使用三角形相似更容易，因为这是一个 2D 问题
// 如果视角为 90 度，问题将更简单，但这里假设不是

// 根据上裁剪面和下裁剪面检测包围球上最上边和最下边的点
float z_test = (0.5)*cam->viewplane_height*sphere_pos.z/cam->view_dist;

if ( ((sphere_pos.y - bhv_tree->radius.y) > z_test)  || // 位于视景体上面
   ((sphere_pos.y + bhv_tree->radius.y) < -z_test) )  // 位于视景体下面
  {
  // 整个节点被剔除，需要设置节点中每个物体的剔除标记
  for (int iobject = 0; iobject < bhv_tree->num_objects; iobject++)
    {
    SET_BIT(bhv_tree->objects[iobject]->state, OBJECT4DV2_STATE_CULLED);
    } // end for iobject

  // 该节点被访问过，且被剔除
  bhv_nodes_visited++;
```

```
    return(1);
    } // end if

  } // end if

  // 至此，我们知道该 BHV 节点没有被剔除
  // 因此遍历其子节点，看能否剔除它们
  for (int ichild = 0; ichild < bhv_tree->num_children; ichild++)
    {
    // 递归调用
    BHV_FrustrumCull(bhv_tree->links[ichild], cam, cull_flags);

    // 可以跟踪被剔除的物体数，如果所有物体都被剔除，则结束

    } // end ichild

  // 指出没有剔除任何物体
  return(0);

  } // end BHV_FrustrumCull
```

函数 BHV_FrustrumCull() 按前面概述的方式遍历 BHV 树，并应用 Cull_OBJECT4DV2() 采用的根据视景体对包围球进行剔除的算法。基本上，我们根据 BHV 节点的球心和半径来创建一个包围球，然后检测该包围球是否在视景体内。

警告：在包围体为球体时，使用包围球很不错；但在包围体为立方体的默认情况下，这样做将降低物体被剔除的步伐。这种问题很容易解决：只需编写一个支持立方体的剔除函数。这项工作留给读者去完成，因为根据如何使用 BHV 技术，可能需要很多不同形状的包围体，但对演示而言，使用球体是最容易的。

提示：实现立方体剔除很简单：只需使用包围立方体的 8 个顶点而不是球体的 6 个切点来进行剔除检测即可。

运行函数 BHV_FrustrumCull() 后，物体数组的 state 字段将被修改（剔除位被设置为 0 或 1）。然后，在物体渲染循环中，对物体数组进行检测，进而忽略被剔除的物体。对于没有被剔除的物体则像以往那样进行处理，其中包括进行标准包围球剔除。

图 13.69 是演示程序 DEMOII13_5.EXE|CPP 运行时的屏幕截图。它看起来并不那么激动人心，因为所有的工作都是在幕后完成的，但屏幕上显示的状态和信息很有趣。这些信息包括被剔除的物体数目、被访问的 BHV 节点数以及进入渲染循环的物体数（经过 BHV 剔除后留下的物体数）。要在启用/禁用 BHV 剔除之间切换，可按 B 键。

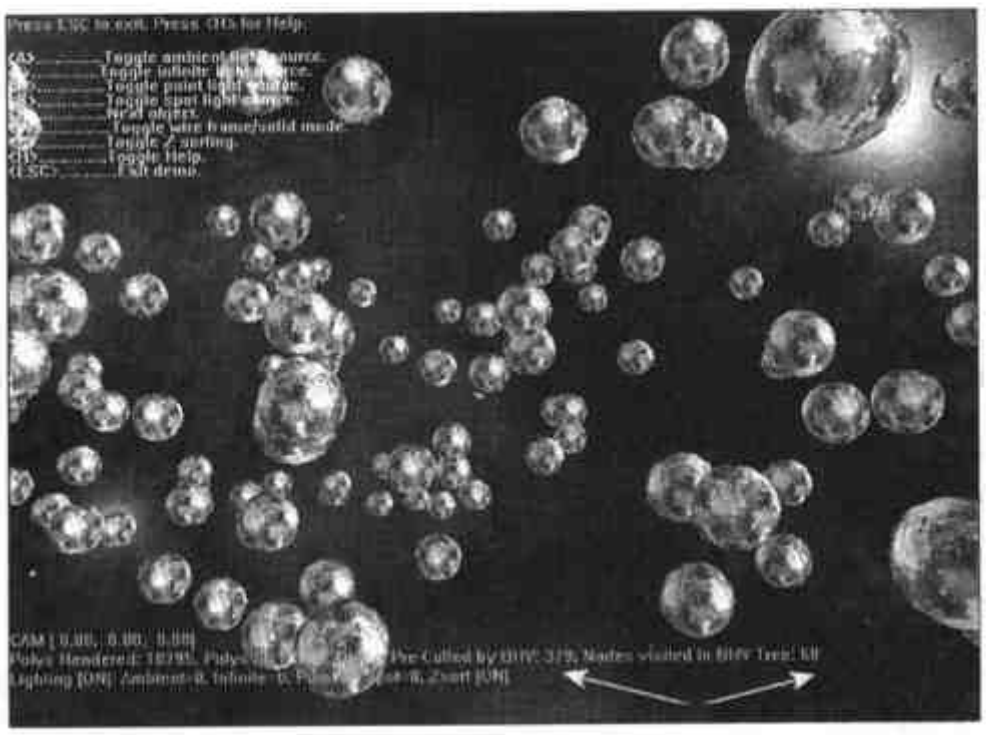


图 13.69　BHV 演示程序的屏幕截图

注意：要编译该演示程序，需要 DEMOII13_5.CPP、所有的库模块（T3DLIB1-11.CPP|H）以及 DirectX .LIB 文件。另外，还需要用作菜单等的资源文件，该文件名为 DEMOII13_5.RC。

13.6.5　八叉树

八叉树是我们刚开发的通过BHV系统的特例。事实上，当参数num_devisions时，函数BHV_Build_Tree()创建的就是八叉树。之所以先编写该函数的代码，是因为它更通用。

这里重申，八叉树是一种特殊的BHV树：以平行于坐标轴的平面将每个BHV节点划分成8个子节点。每个节点的大小相同，形状为立方体。前面的BHV系统支持诸如球体等非立方包围体，因为我们在每个节点中，都使用一个向量来存储半径，同时有一个attr字段可用于描述包围体的类型。

因此，我们已经有了八叉树系统，无需编写新的代码。然而，有关生成八叉树，有几点需要说明。

八叉树（和BHV树）可用于物体或多边形。我们将其用于物体而不是多边形（前面将BSP用于多边形，因此这里将BHV用于物体），因为对于包含数千个物体的大型室外世界来说，八叉树和BHV树确实很有用。

在3D空间中，八叉树的每个节点最多有8个子节点，然而并没有要求一定要有8个子节点。

例如，图13.70说明了八叉树BHV算法进行空间划分时的几次迭代（2D投影），其中的数字表示迭代次数。从图中可知，仅当节点包含物体或多边形时，才进一步对其进行划分，这样可以节省内存。如果读者愿意，完全可以在每次迭代时都将每个节点划分成8个，在有些节点中不包含任何物体；但作者喜欢仅当节点包含物体时才创建它，否则将其设置为NULL，前面的函数BHV_Build_Tree()就是这样做的。

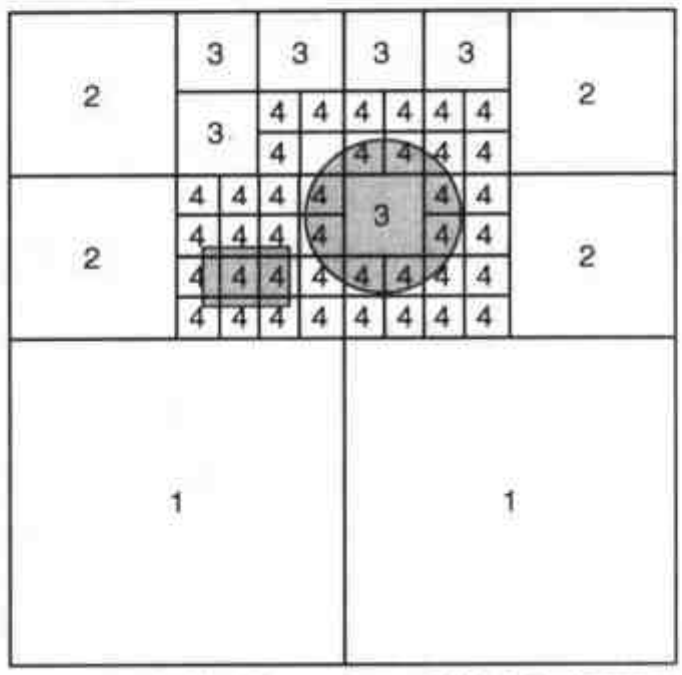


图 13.70　八叉树划分的2D表示（从技术上说是四叉树）

像BHV树一样，创建八叉树时也有很多终止条件。可能在节点的大小达到特定的值时终止，也可能在达到特定的层数后终止，还可能在每个节点包含的多边形数目达到特定的值后终止。选择权在读者手中。

总之，八叉树是一种特殊的BHV树。要创建八叉树，只需调用函数BHV_Build_Tree()，并将参数num_divisions设置为2即可，如下所示：

```
BHV_Build_Tree(&bhv_tree,
        scene_objects,
        NUM_SCENE_OBJECTS,
        0,
        2);
```

根据视景体对八叉树进行剔除的方式与前面介绍的完全相同。使用八叉树而不是其他划分方式（如沿每条轴每个节点划分成10份，得到1000个球体）的惟一原因是，它很好地模拟了3D坐标系统由8个象限组成的特性，具有一些很好的特性，更容易处理。

13.7　遮 掩 剔 除

本章要讨论的最后一个主题同样与绘制可见的多边形相关。我们知道，使用PVS技术，可以生成从游戏中任何视点可见的多边形列表，然后再根据这些列表来确定在某一帧中需要考虑哪些多边形。然而，如果不使用PVS，或者由于游戏世界的类型使得PVS不可行或不管用，该如何办呢？

还有一种名为遮掩剔除（occlusion culling）的技术，它从另一个角度来解决这种问题。实际上，不想使用有关可见性的静态解决方案时（游戏世界可能频繁变化，PVS、BSP和八叉树不适合，但有些重要的几何元素，它们很大，会遮掩大部分场景），可使用遮掩剔除技术。

例如，请看图13.71所示的Unreal 2003的屏幕截图。玩家转身后的屏幕截图如图13.72所示。转眼之

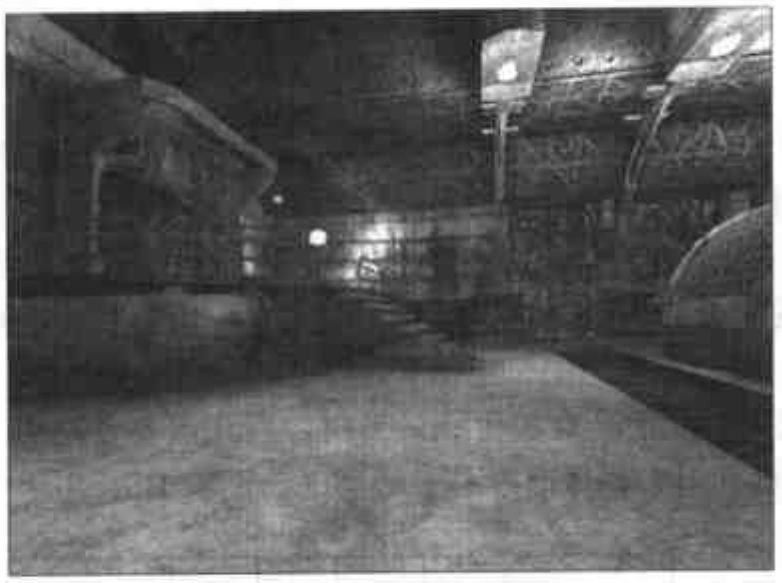


图 13.71　Unreal 2003 中一种视野开阔的情形

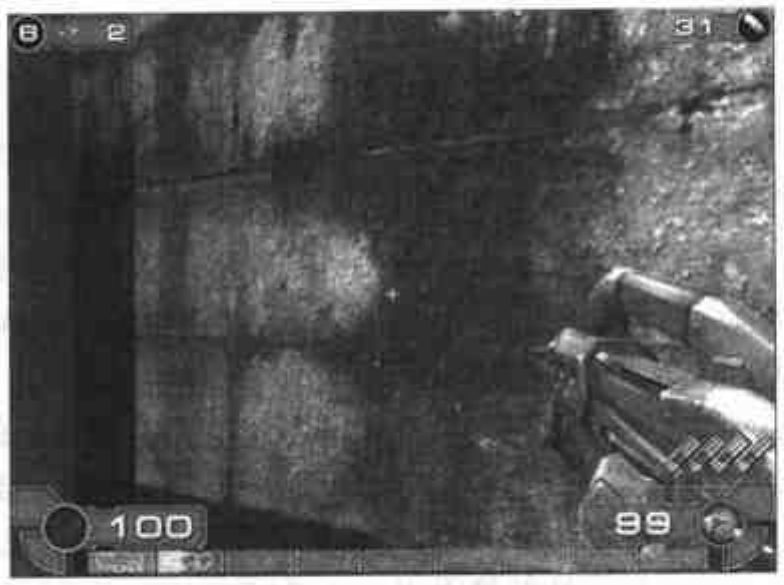


图 13.72　玩家转身后的情形

间，玩家的视线就被一个大型物体遮住；在这种情况下，PVS 不管用（除非基于视角来创建 PVS，但这样做 PVS 将非常大）。入口技术也不管用，但必须采用某种技术来处理这种极其简单的情形，以便不渲染甚至不考虑被物体遮住的多边形。这就是遮掩剔除的基础。

13.7.1　遮掩体

将这种问题具体化，看一看包含城市风景的室外游戏中的情况。图 13.73a 是一个城市风景（实际上是波士顿）的俯视图，其中的小方块表示建筑物网格。这里没有房间、入口等。本可以使用 PVS，但由于没有室内几何体，必须将视景体作为包围单元格来创建每个视点的 PVS。还是将这种技术忘了吧，将注意力放在显而易见的东西上。请注意图 13.73a 中的视点和视景体以及视点前面的大型黑色建筑；在来看图 13.73b，从中可知，该建筑物遮掩了 90%可能可见的多边形。

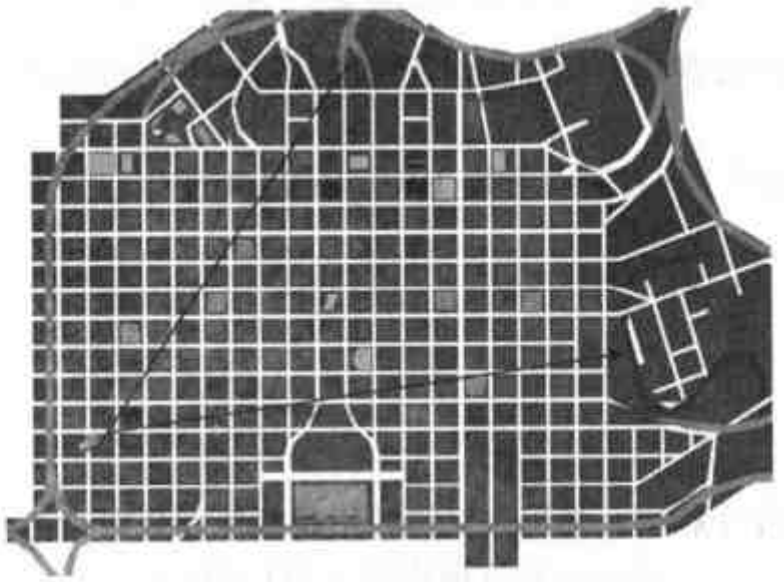


图 13.73a　用于说明遮掩的城市风景

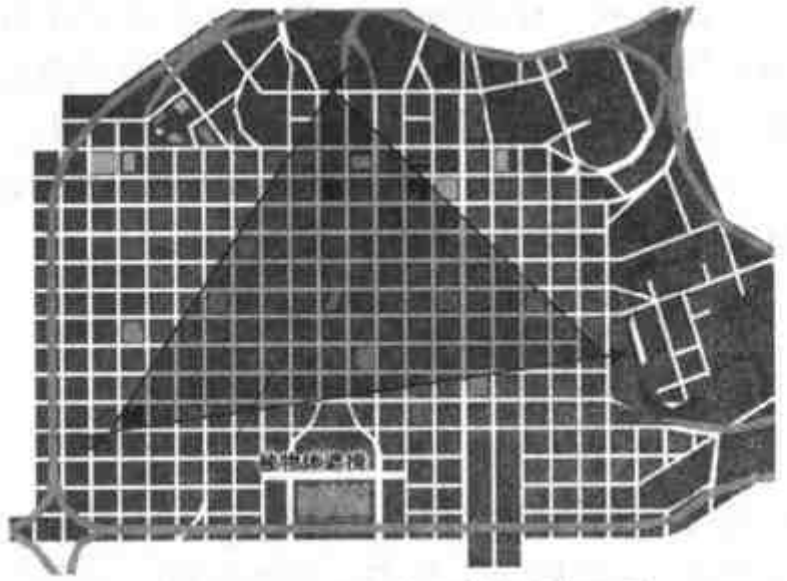


图 13.73b　包含遮掩体的城市风景

因此，如果创建一个遮掩体（occlusion volume），位于该遮掩体内的物体都不可见，便大功告成。当然，创建遮掩体通常并不容易。有很多可用于找出 3D 物体轮廓边界的算法，根据轮廓，可以将很多面进行投影，这些投影定义了一个遮掩体。

位于遮掩体内的一切都被剔除。可以进一步深化这种思想，根据多个遮掩物（occluder）创建多个遮掩体。当然存在一个回报减少点：判断多边形是否位于遮掩体内的时间比绘制多边形还长。然而，并没有规定只能将多边形作为最小的遮掩判断单位，可以对物体、BPS 树、八叉树节点等进行检测，看其是否被遮掩。

13.7.2　选择遮掩物

要使用遮掩剔除技术，接下来的一步是选择遮掩物。需要使用某种方法来选择潜在的遮掩物，甚至可以预先进行处理，计算包围遮掩物的轮廓。用于选择遮掩物的策略很多，它们分为两大类：离线遮掩物选择和运行阶段遮掩物选择。一种离线遮掩物选择策略是，在建模程序中将某些物体标记为潜在遮掩物，这

些物体很大或者在玩家玩游戏时会经常遮住玩家的视线。另外，还可能基于每个分区，选择一组潜在的遮掩物。

以在线方式选择遮掩物时，通常包含一个离线选择阶段——至少离线标记一些潜在的遮掩物。在运行阶段，系统将做出这样的判断：将潜在遮掩物加入到遮掩列表，进而将遮掩列表转换为遮掩体，并据此进行裁剪/剔除是否值得。试探方法可能基于每个遮掩物的大小，这意味着只有移动的大型物体、大型地形或大型环境物体才被标记为潜在遮掩物。然后，在运行阶段，系统将潜在遮掩物投影到屏幕空间中，并判断将其用作遮掩物是否值得。接下来介绍一种作者喜欢使用的混合选择方法。

13.7.3　混合型遮掩物选择方法

可以在离线处理或建模阶段选择遮掩物；也可以在运行阶段从一组潜在遮掩物进行选择；还可以动态地进行选择：计算大型物体（多边形）的投影面积，然后判断将其用作遮掩物是否值得。要支持所有这些技术，需要在建模和渲染流水线中做些工作。然后在很多情况下，可以以简单的方法来基于多边形或物体选择遮掩物，其步骤如下：

1. 遍历完全位于视景体内且可能可见的所有物体，选择 n 个最近、最大的物体。

2. 将物体投影到屏幕空间中，并计算投影区域的面积。从中选择 $m(<=n)$ 个投影面积最大的物体。然后计算每个遮掩物的内接矩形（虽然这将导致某些本被遮掩的多边形漏网，但任意多边形或网格相比，对矩形进行投影容易得多）。

3. 将遮掩矩形加入到遮掩体中，然后根据遮掩体对所有物体和多边形进行检测，剔除所有被遮掩的几何体。

实际上，这种算法非常简单：快速选择几个最近、最大的物体或几何体，将其作为潜在遮掩物；然后将它们投影到屏幕空间中，并从中选择一组可能遮掩很大视野的遮掩物。然而，创建遮掩体时，不使用复杂的几何形状，而是使用完全位于每个遮掩物内的内接矩形。然后，对于任何比遮掩物平面远且位于遮掩体内的物体都不予考虑。当然，在以某种层次方式（如 BSP、房间、八叉树等）表示物体/游戏世界时，这种方法的效率最高，因为这样可以一次性判断出大量物体被遮掩，进而不予考虑它们。

13.8　总　　结

这一章非常棒，介绍了所有重要的空间划分算法，揭示了其中很多算法的本质，并实现了最重要的几种算法。有了这些知识后，读者可以混合使用它们，创建出自己的算法。事实上，读者将发现，有关该主题，自己拥有的知识已经绰绰有余。

然而，开始编写任何算法之前，读者务必阅读手边有关该主题的所有资料，以免做过多的重复工作——几乎所有的事情都有人尝试过。当然，正如读者知道的，作者提倡亲身去体验各种东西，但如果您时间有限，仔细阅读有关该主题的资料是值得的。

最后，需要警告读者的是，很多算法都是递归的，因此编程时一定要小心，因为递归程序的调试工作非常可怕。

第 14 章　阴影和光照映射

本章介绍一些可提高场景真实感的技术：阴影和光照映射。和以往一样，本章将详细讨论如何使用软件来实现这些技术，包括以下内容：

- 新的游戏引擎库模块；
- 概述；
- 有关阴影的物理学；
- 使用透视图像和广告牌模拟阴影；
- 平面网格阴影映射；
- 光照映射和面缓存技术。

14.1　新的游戏引擎模块

同样本章的函数很多，可将其作为一个独立的库模块。该库模块名为 T3DLIB12.CPP|H，因此要编译本章的程序，需要程序的主.CPP 文件、DirectX 文件以及下述新的库模块：

- T3DLIB12.CPP——支持阴影和光照映射的 C/C++源代码；
- T3DLIB12.H——头文件。

当然，还需要链接库模块 T3DLIB1-11.CPP|H。

14.2　概　　述

之所以推迟到现在才介绍本章的内容，旨在先介绍足够的技术、演示程序和技巧，让读者具备实现这些技术的技能。当然，作者的意思是说，之前已经或多或少地介绍过光照映射，而不是要将读者丢下不管。第 8 章介绍过如何混合纹理和光照效果；至于阴影，我们编写了支持 alpha 混合的光栅化函数，可用于绘制阴影投影。因此，本章要做的工作是，将已有的函数进行系统集成，并添加两项新技术，以便能够编写演示程序。在很大程度上说，读者将能够完成上述工作，因为读者已经知道如何完成这些工作。

为介绍本章的主要主题，最佳的方法是从与阴影相关的物理学着手，然后实现两种阴影技术（这样的技术还有很多，但它们更适用于硬件引擎，还是留给有关 Direct3D/OpenGL 的图书去介绍吧）。从介绍最简单的光照映射和面缓存（surface caching）技术开始，但不在引擎中加入对这种技术的支持。作者之所以做出这样的决定，是因为在引擎中支持光照映射所需的代码是简单范例的 10 倍。如果读者像作者一样，将会认为简单范例比冗长的实现更有意义。

本书带领读者进行的 3D 游戏编程之旅已接近尾声，读者有能力创建能够渲染类似于 Quake/Unreal 游戏世界的软件引擎。阅读本章后，读者将知道如何实现更多的特殊效果，从很大程度上说，读者已经为此做好了准备。

14.3　简化的阴影物理学

有关光子及其如何与表面交互的细节远远超出了模拟阴影的需求范围，但我们至少需要知道如何生成阴影以及它们为何有软阴影和硬阴影，本书前面讨论光照时，将其称为本影（umbra）和半影（penumbra）。

在图 14.1 中，一个圆柱放置在一个平面上，圆柱上面有一个点光源。另外，还有微弱的环境光，因此，无论从什么方向看，圆柱都是可见的。同时圆柱在地面上投下了一个阴影，这是因为它遮住了点光源照射到地面上的部分光线。如果读者仔细查看渲染结果，将发现这导致了一种有趣的二级效应（second-order effect）。

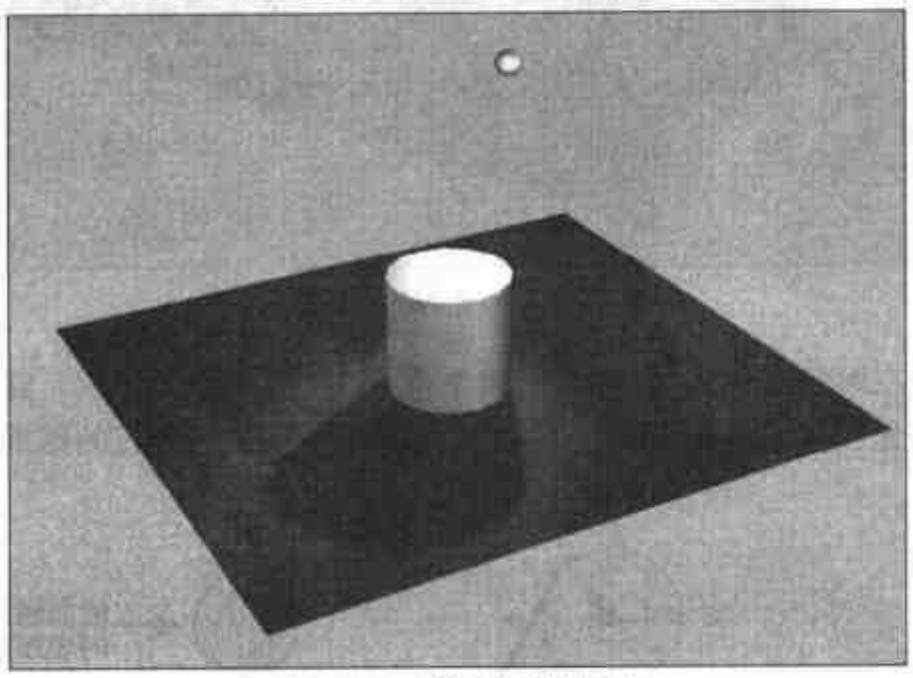


图 14.1　圆柱投下的阴影

通过分析阴影本身，读者将发现，在阴影的中间有一个黑色区域（本影），但在边缘附近阴影开始减弱，形成一个较亮的区域（半影）。另外，半影只出现在远离圆柱底座的阴影边界处，在圆柱底座下则没有。

在圆柱的底座下，阴影非常强，这是我们需要在引擎中模拟（实现）的细节。下面从光子的角度讨论发生的情况。

光线跟踪和强度计算

图 14.2a 是上述圆柱范例的侧视图，其中有一个点光源。通过跟踪光源照射到地面的光线，将发现结果非常简单：点光源将照亮不位于（被圆柱遮住的）阴影区域内的一切。因此，得到如图 14.2b 所示的硬阴影。

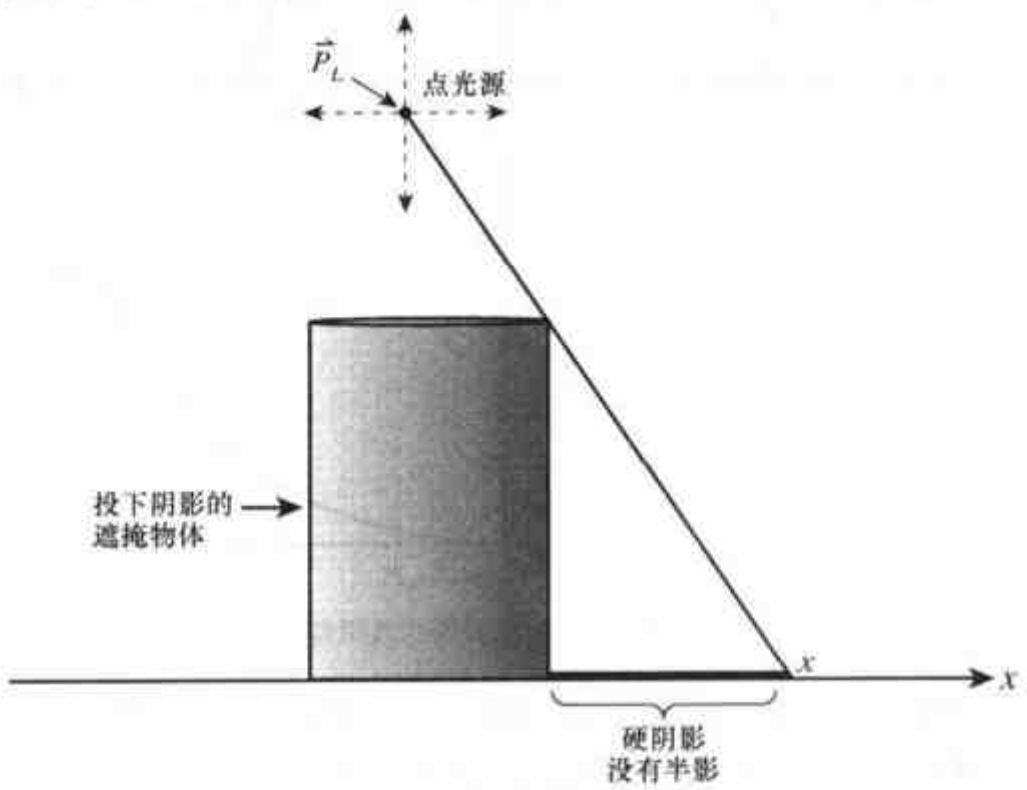


图 14.2a　点光源的光线跟踪

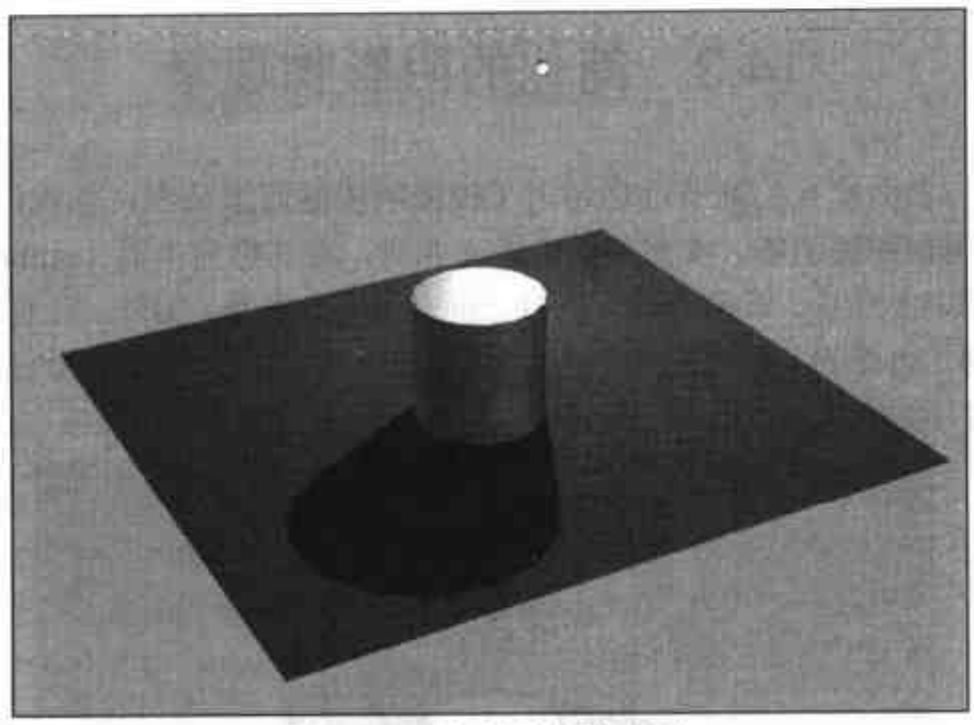


图 14.2b　点光源的光线跟踪

如果光源不是点光源，而是球形光源（如图 14.3 所示），情况将如何呢？在这种情况下，在圆柱投下的阴影的 x 轴上，各点的光照强度将不相同：与球形光源上可照射到该点的球面的面积成正比。

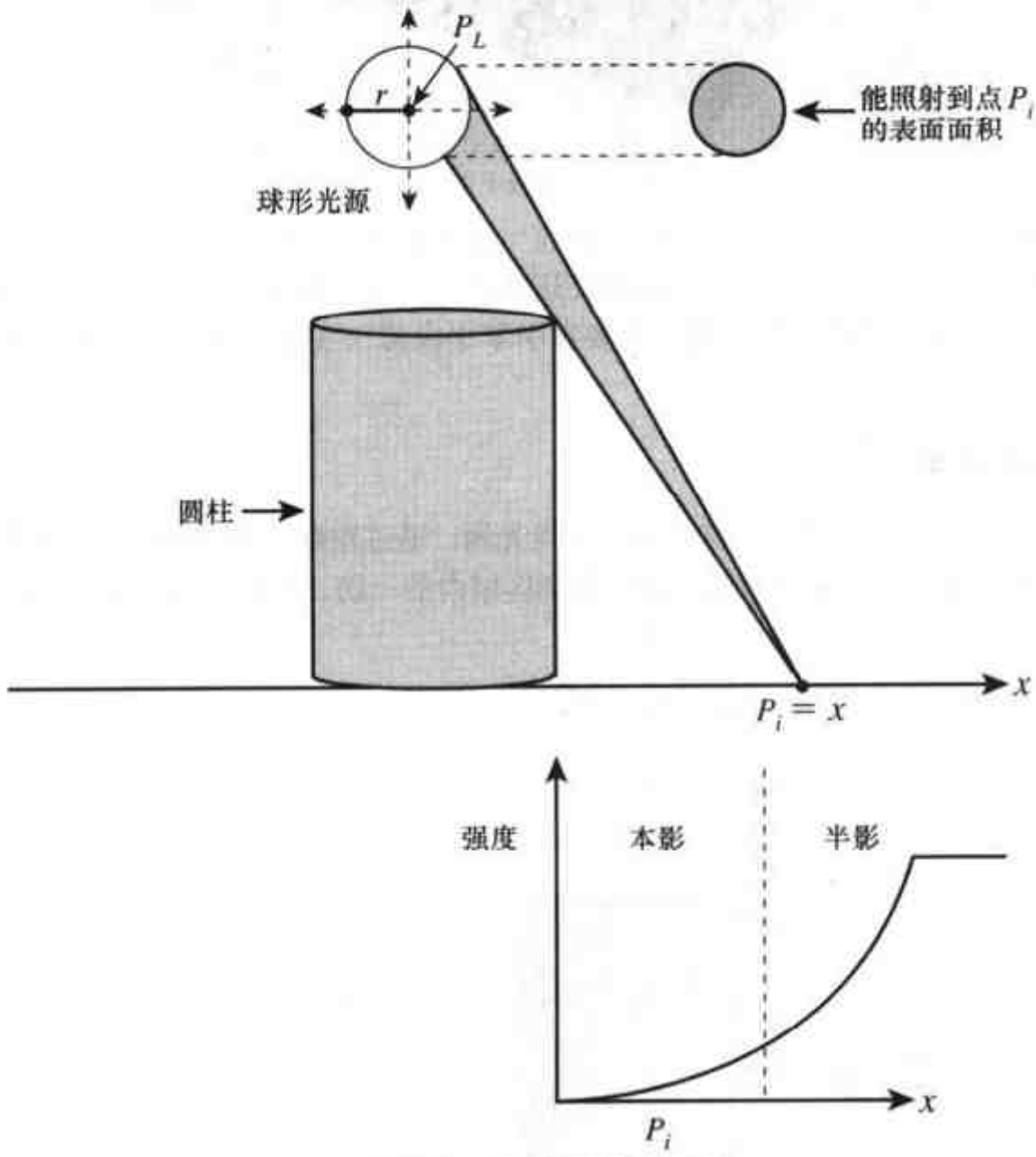


图 14.3　球形光源的光线跟踪

接收点的光照强度与球形光源上可照射到该点的球面的面积成正比，沿 x 轴移动时，这种面积增加或减少，从而形成半影。向阴影边缘移动时，能照射到测试点的光源表面面积将增加，因此光照强度增加（阴影减弱），图 14.4b 是这种情形的渲染结果。最后，球形光源的尺寸增大时，阴影将减弱，如图 14.4a 和 14.4b 所示。

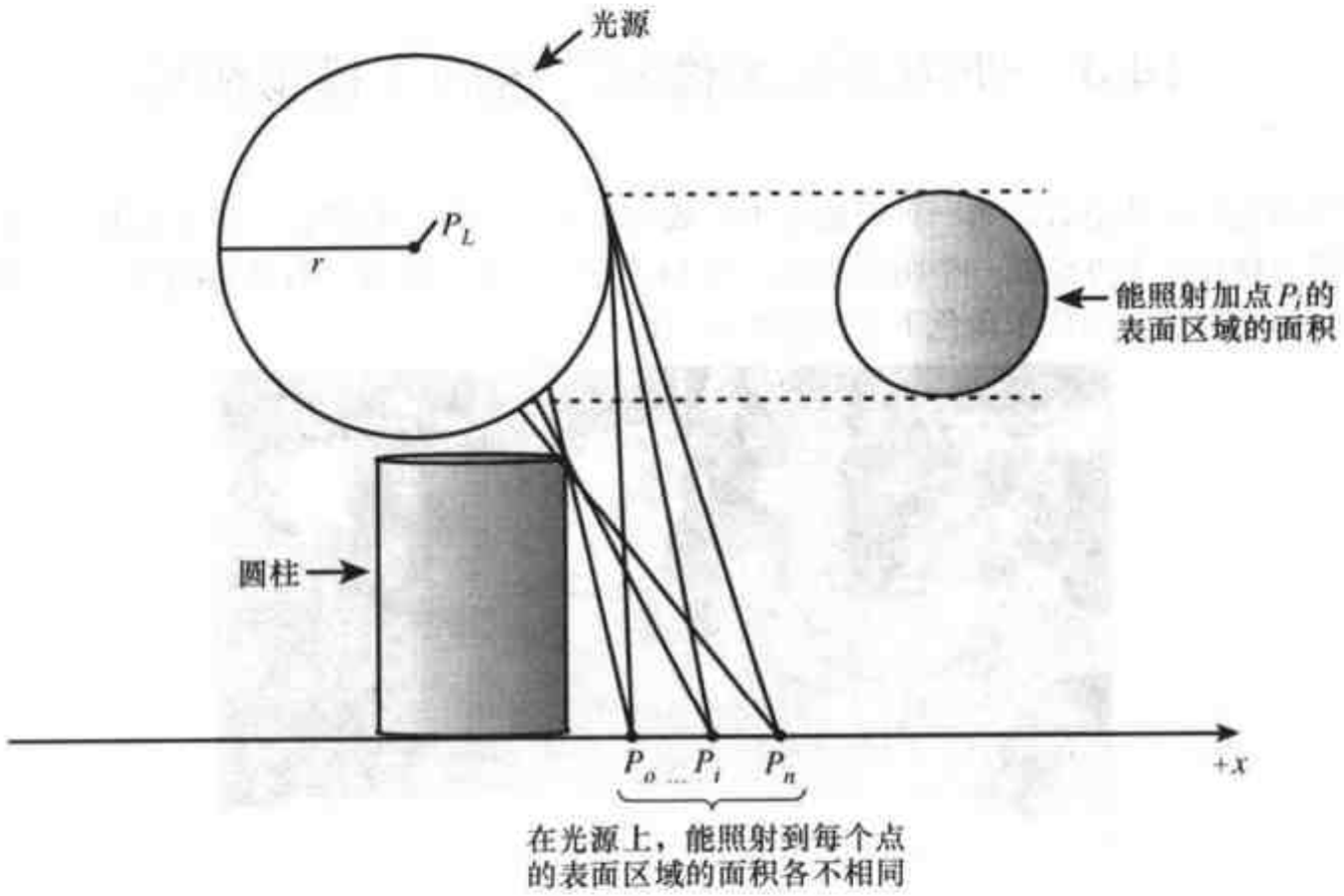


图 14.4a　点光源的尺寸越大，阴影越弱

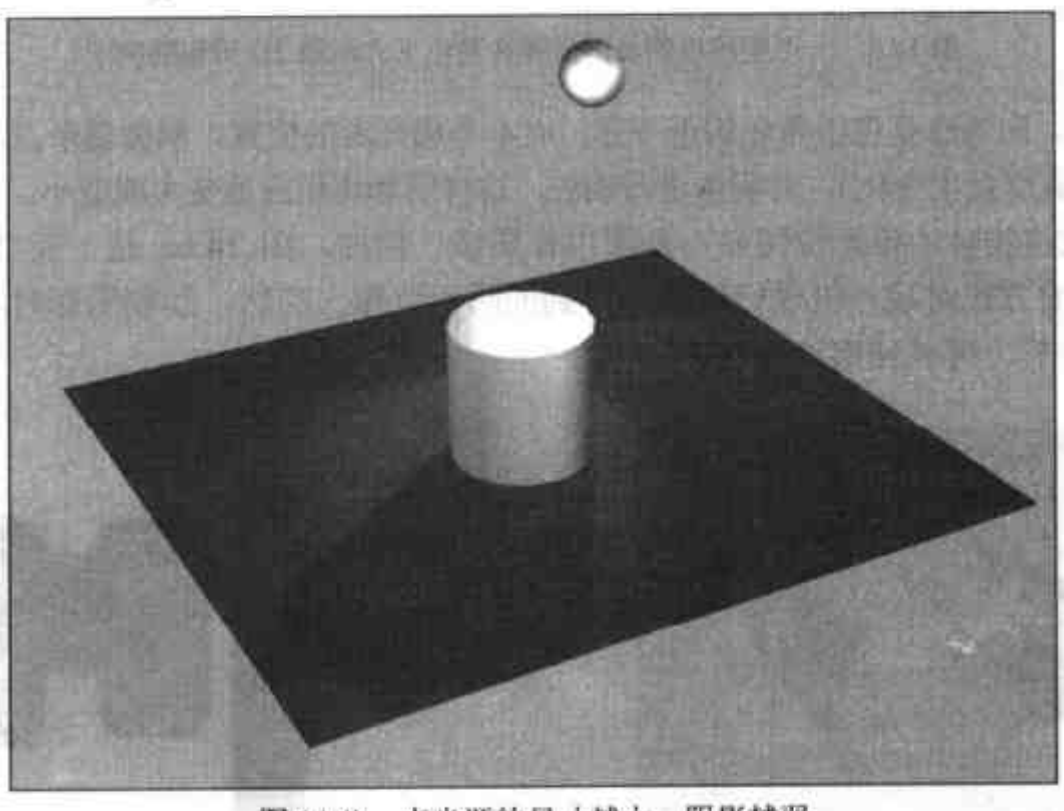


图 14.4b　点光源的尺寸越大，阴影越弱

因此，要正确地模拟本影和半影，必须考虑光源的大小，还要做一些光线跟踪工作。然而，这样做的计算量非常大。当然，并不是说不能这样做，但有必要吗？答案是否定的。编写本书时，很多游戏并不支持阴影；当然也有很多游戏支持，但使用的是硬阴影，没有半影。然而，有些游戏确实使用了软阴影，效果非常好（通常是使用多重纹理映射、像素 Shader 和模板缓存实现的）。

重要的是，在将近 1/4 世纪内，一直有不支持阴影（从物理学上说正确的阴影）的视频游戏，因此有阴影总比没有阴影强。虽然我们不打算计算每个像素的光照强度，但可以使用 alpha 混合或其他技巧来创建软阴影。对软件引擎而言，这足够了。

读者从光学的角度知道阴影、本影和半影是如何形成的后，接下来介绍如何创建阴影——从简单阴影到真实感相当高的阴影。

14.4 使用透视图像和广告牌来模拟阴影

就遮光物的几何形状而言，第一种阴影技术生成的几乎不是真正的阴影，但效果非常不错。这里说的是在生成阴影的物体正下方绘制一个阴影位图。图 14.5 是一个角色物体下有透视阴影（广告牌）的典型范例，其中的阴影不过是一个位于角色下方的圆形斑点而已。

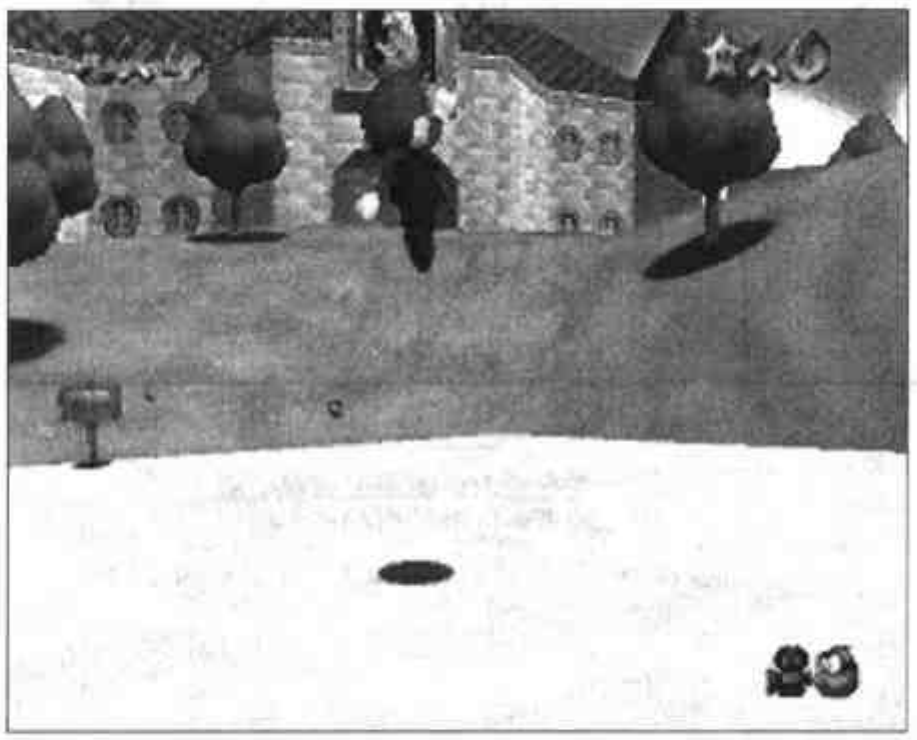


图 14.5 一个使用位图或广告牌在角色下方渲染 2D 阴影的例子

最简单的情形是，阴影总是位于角色的正下方，而不考虑光源的位置。稍微复杂点的情形是，当角色沿 y 轴上下移动时（物体高度发生变化），对阴影进行缩放，这样阴影将相应地变大或变小。这种技术也很常见。

另外，很多游戏将物体的投影位图而不是圆用作阴影。例如，图 14.6a 是一架 3D 太空战斗机，而图 14.6b 是以从上往下的方式对战斗机进行投影时得到的阴影位图（当然，绘制阴影时将调整位图的大小）。如果要增加阴影的细节，可以使用这种位图。

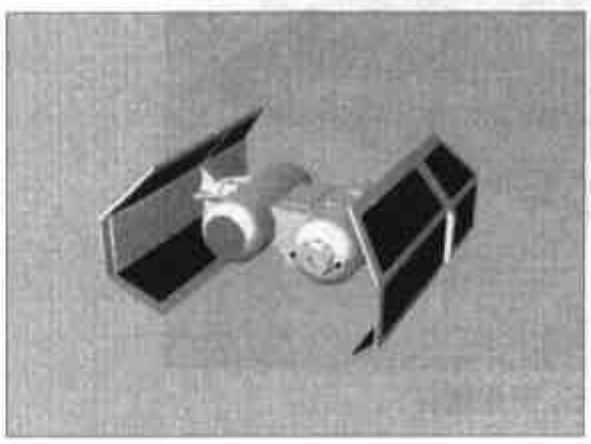


图 14.6a 太空战斗机

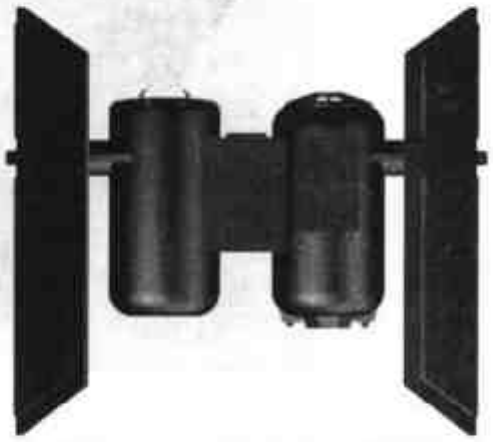


图 14.6b 战斗机的 2D 投影位图，将用作广告牌阴影纹理

然而，这种技术存在问题：对于每个物体，都需要一个阴影位图。这没什么，但阴影为物体的投影位图意味着与朝向相关，当物体旋转时，位图也必须跟着旋转。如果将位图作为纹理映射到多边形广告牌上（如图 14.7 所示），将可以解决这种问题，因为可以旋转广告牌，后者将相应地旋转纹理。然而，刚开始接触阴影时，最好还是对所有的物体都使用同一个阴影位图。以后再尝试对每个物体使用不同的阴影位图，并跟踪物体的朝向变化，进而对阴影进行合适的旋转或变换。

在阴影模型中，可以实现的更为复杂的技术是，不但根据物体的高度改变阴影的大小，并考虑某个光源的影响：光源将沿什么方向照射物体，进而将阴影投射到地面上。图 14.8 简要地说明了这种理念。同样，这也简单，因为我们将根据物体中心和光源中心来确定投影线与地面的交点，然后在交点处绘制一个圆形

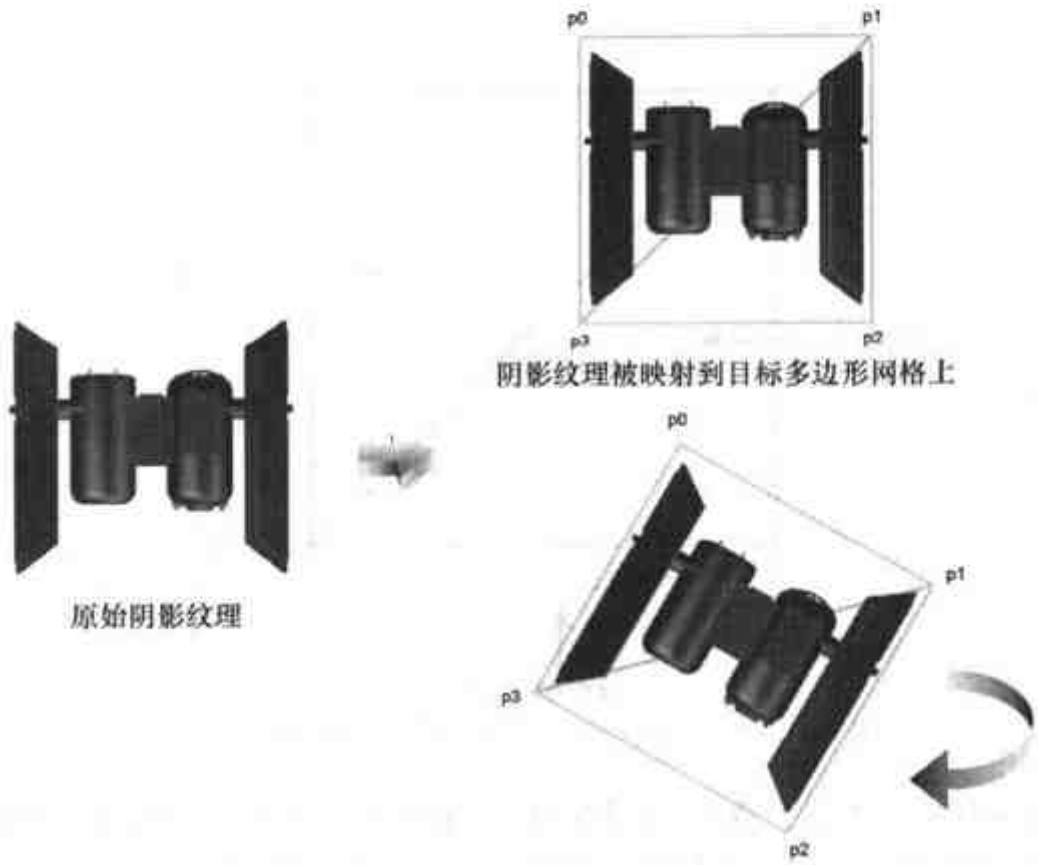


图 14.7　将阴影纹理映射到多边形网格上，以简化旋转

阴影。虽然如此，其效果还是相当不错的。惟一遗漏的内容是，没有对阴影进行错切（shearing）——我们总是绘制同一个位图（可能对其进行了旋转）。下一节介绍平面阴影时，将实现错切功能；现在处于简化的目的，暂时不去管它。

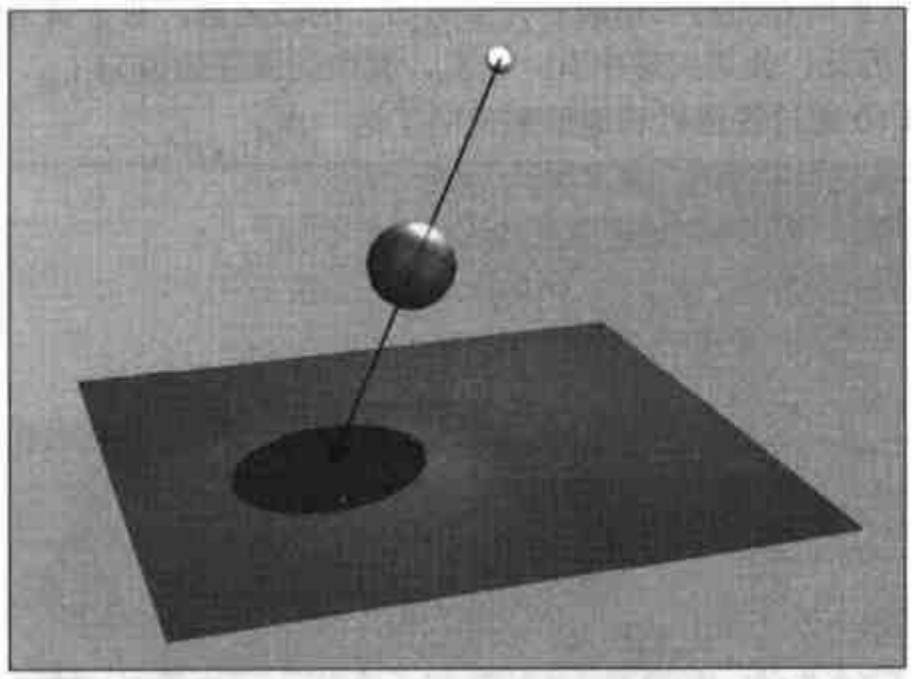


图 14.8　对阴影进行缩放并通过投影来确定其在地面上的位置

现在要做的是，实现前面介绍的每个理念。为此，必须能够绘制广告牌（方形多边形），将指定了透明值（transparent value）的纹理映射到广告牌上。实际渲染阴影时，需要使用这种功能。我们的计划是，使用两个三角形来创建一个正方形，然后将一个用作阴影的黑色圆形区域作为纹理，将其映射到正方形上。然而，在纹理的透明像素处，必须是透明的。这是一个问题，因为我们的光栅化函数还不支持透明功能。因此，必须重新编写某些光栅化函数，加入对这种功能的支持。

14.4.1　编写支持透明功能的光栅化函数

我们几乎为绘制阴影（广告牌）准备就绪。我们将创建一个由两个多边形组成的网格，如图 14.9 所示；然后将阴影纹理映射到该网格上。惟一的问题是，当前所有的纹理映射函数都只能处理不透明的纹理——通过纹理的任何一部分都无法看到后面的物体。

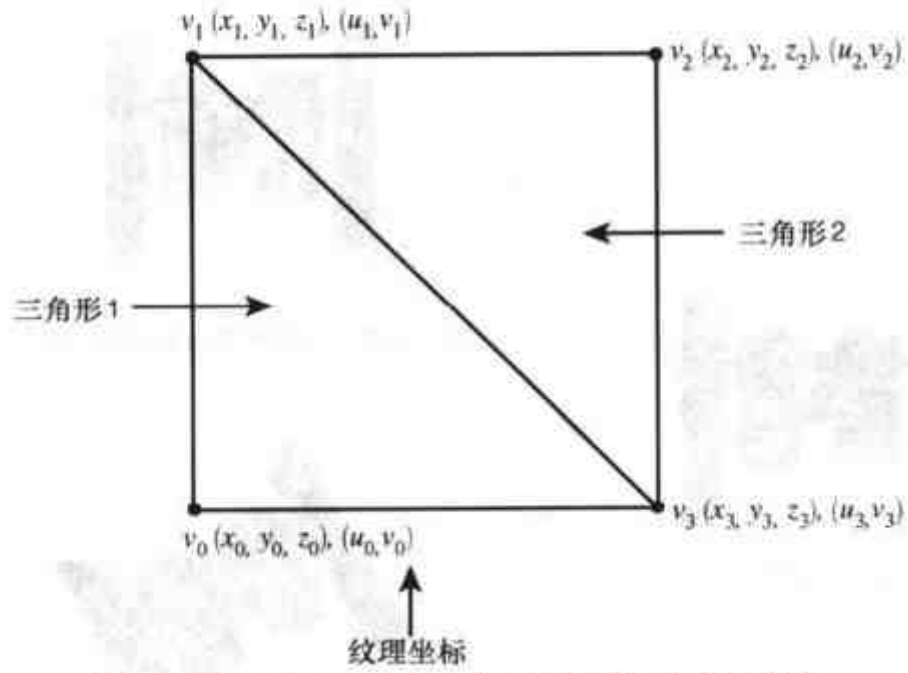


图 14.9　基于由两个多边形组成的简单网格的广告牌

提示：在 3D 游戏编程中，术语广告牌（billboard）指的是一个其上被贴上 2D 图像的多边形，该多边形通常与视平面或其他平面平行。广告牌技术（billboarding）用于在 2D 平面上绘制 3D 物体的 2D 投影，使其看起来很真实。在很多 3D 游戏中，使用广告牌来表示树木、房屋或其他无需用 3D 多边形网格来表示的物体（焰火和爆炸非常适合用广告牌来表示，它们不是由多边形组成的，而是始终面对观察者的 2D 图像）。我们以简单的方式实现广告牌，以便在上面绘制阴影，然后在地面上绘制带阴影纹理的广告牌。

在现有的光栅化函数中增加透明支持非常简单。然而，有必要统一一下作者和读者的认识：这里所说的透明并非 alpha 混合。这里所说的透明指的是完全透明，也就是说，对于值为指定值（i_{rgb}）的像素，不对其进行渲染（光栅化）。因此，如果纹理中有一个孔，其中的像素的值为 i_{rgb}，则通过这个孔，将能够看到纹理后面的东西。图 14.10 通过铁栅栏和地形来说明了这一点。

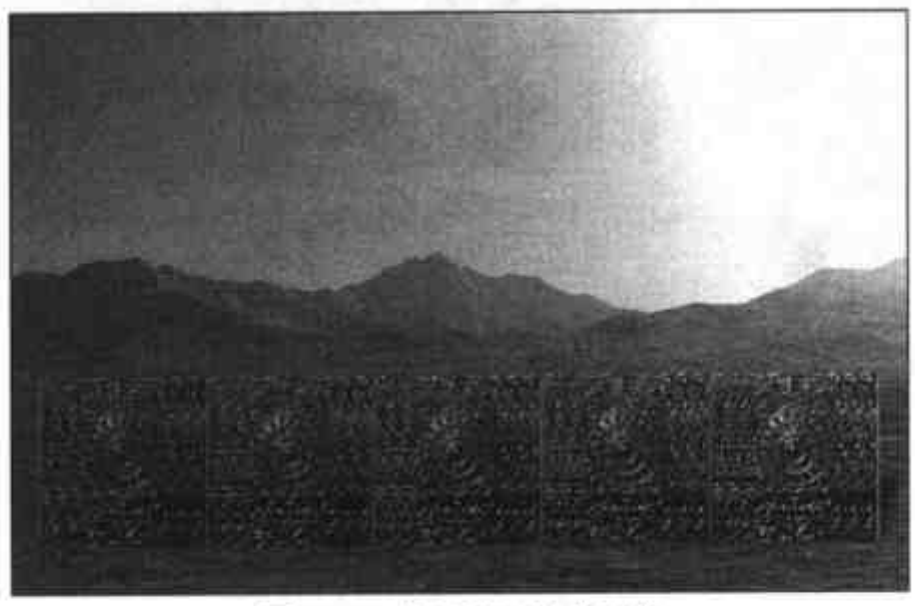


图 14.10　透明纹理映射示例

作者视速度如命，不打算使用普通值来表示透明颜色，而是使用 RGB 值（0, 0, 0），这样，在纹理中，透明像素的 16 位颜色值为 0。因此，只需在原来的光栅化函数的内循环中加入这样的比较逻辑即可：

```
if (pixel)
  {
  // 渲染像素
  } // end if
```

如果要使用否定逻辑，可以这样做：

```
if (!pixel)
  {
  // 不绘制
  } // end if
```

无论采用哪种方式，0 都是一个很好比较的数字，使用它可以进行大量的优化。将 RGB(0，0，0)用作透明颜色的惟一缺点是，它表示的是黑色。如果纹理中有黑色像素，而我们又希望它完全是不透明的，该如何办？将其改为（0，0，1）或（1，1，1）——肉眼将看不出差别。虽然有些人将 RGB(255，0，255)（粉红色）用作透明色，但我们使用黑色。

这里不打算重新编写每个光栅化函数，因为透明功能用于阴影和光照，而对于阴影而言，无需使用高级着色。然而，为避免过于肤浅，将重新编写足够多的光栅化函数，至少让读者使用 z 缓存进行渲染时，能够使用透明纹理。如果读者要在使用 1/z 缓存的光栅化函数中支持透明纹理，可以自己修改它们（修改一个光栅化函数需要大约 5 分钟）。介绍新的函数原型之前，先来看一下支持透明纹理、使用固定着色的仿射纹理映射函数的主循环，该函数将颜色为 RGB(0，0，0)像素视为透明的：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  textel = textmap[(ui >> FIXP16_SHIFT) +
          ((vi >> FIXP16_SHIFT) << texture_shift2)];

  //检测像素是否是透明色
  if (textel)
    {
    // 更新 z 缓存
    z_ptr[xi] = zi;
    // 绘制像素
    screen_ptr[xi] = textel;
    } // end if


  // 计算下一个像素的 u、v 和 z
  ui+=du;
  vi+=dv;
  zi+=dz;
  } // end for xi
```

其中新增的代码为粗体，只有一行。为方便读者比较，下面是加入透明支持前该函数中相应的代码：

```
// 绘制扫描线
for (xi=xstart; xi < xend; xi++)
  {
  // 总是更新 z 缓存
  // 绘制像素
  screen_ptr[xi] = textmap[(ui >> FIXP16_SHIFT) +
              ((vi >> FIXP16_SHIFT) << texture_shift2)];

  // 更新 z 缓存
  z_ptr[xi] = zi;

  // 计算下一个像素的 u、v、z
  ui+=du;
  vi+=dv;
  zi+=dz;
  } // end for xi
```

新的纹理映射函数读取纹素，并将颜色同 RGB(0，0，0)进行比较。如果不是 (0，0，0)，则绘制该纹素，并更新 z 缓存。

提示：纹素为透明时不更新 z 缓存。它们是透明的，因此对 z 缓存没有影响。

14.4.2　新的库模块

新的光栅化函数位于 T3DLIB12.CPP|H 中。下面列出头文件的内容，供读者参考：

```
// T3DLIB12.H: T3DLIB12.CPP 的头文件
```

```
// 防止重复包含
#ifndef T3DLIB12
#define T3DLIB12

// 外部变量 ///////////////////////////////////////////////////////////

extern HWND main_window_handle; // 窗口句柄
extern HINSTANCE main_instance; // 实例

// 原型 ///////////////////////////////////////////////////////////

// 新的渲染场境函数，调用支持透明功能的光栅化函数
void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_3(RENDERCONTEXTV1_PTR rc);

// 使用 z 缓存、直接写入 z 缓存且支持透明功能
void Draw_Textured_TriangleGSWTZB2_16(POLYF4DV2_PTR face, // 多边形面指针
          UCHAR *_dest_buffer,    // 视频缓存指针
          int mem_pitch,          // 每行多少字节（320、640 等）
          UCHAR *_zbuffer,        // 指向 z 缓存的指针
          int zpitch);           // z 缓存中每行多少字节

void Draw_Textured_TriangleFSWTZB3_16(POLYF4DV2_PTR face, // 多边形面指针
          UCHAR *_dest_buffer,    // 视频缓存指针
          int mem_pitch,          // 每行多少字节（320、640 等）
          UCHAR *_zbuffer,        // 指向 z 缓存的指针
          int zpitch);           // z 缓存中每行多少字节

void Draw_Textured_TriangleWTZB3_16(POLYF4DV2_PTR face,  // 多边形面指针
       UCHAR *_dest_buffer,          // 视频缓存指针
       int mem_pitch,     // 每行多少字节（320、640 等）
       UCHAR *_zbuffer,    // 指向 z 缓存的指针
       int zpitch);      // z 缓存中每行多少字节

// 使用 z 缓存、支持透明功能
void Draw_Textured_TriangleZB3_16(POLYF4DV2_PTR face,  // 多边形面指针
       UCHAR *_dest_buffer,  // 视频缓存指针
       int mem_pitch,     // 每行多少字节（320、640 等）
       UCHAR *_zbuffer,    // 指向 z 缓存的指针
       int zpitch);      // z 缓存中每行多少字节

// 使用 z 缓存、直接写入 z 缓存、支持透明功能和 alpha 混合
void Draw_Textured_TriangleWTZB_Alpha16_2(POLYF4DV2_PTR face,  // 多边形面指针
       UCHAR *_dest_buffer,   // 视频缓存指针
       int mem_pitch,     // 每行多少字节（320、640 等）
       UCHAR *_zbuffer,    // 指向 z 缓存的指针
       int zpitch,       // z 缓存中每行多少字节
       int alpha);

void Draw_Textured_TriangleZB_Alpha16_2(POLYF4DV2_PTR face,// 多边形面指针
       UCHAR *_dest_buffer,   // 视频缓存指针
       int mem_pitch,      // 每行多少字节（320、640 等）
       UCHAR *_zbuffer,     // 指向 z 缓存的指针
       int zpitch,        // z 缓存中每行多少字节
       int alpha);

void Draw_Textured_Triangle_Alpha16_2(POLYF4DV2_PTR face,   // 多边形面指针
       UCHAR *_dest_buffer,     // 视频缓存指针
       int mem_pitch, int alpha); // 每行多少字节（320、640 等）

#endif
```

在头文件中，定义了 6 个新的光栅化函数，除新增了对透明功能的支持外，它们与前一个版本相同。作者重新编写了足够多的函数，让读者能够在使用 z 缓存或不使用深度缓存的情况下，以 alpha 混合和透明的方式绘制纹理。正如读者知道的，我们早已不再手工调用光栅化函数了，而是先设置渲染场境，然后调用渲染场境函数。这个函数的前一个版本为 2.0，现在为 3.0：

```
void Draw_RENDERLIST4DV2_RENDERCONTEXTV1_16_3(RENDERCONTEXTV1_PTR rc);
```

除了名称中包含数字“3”外，这个函数与前一个版本没有任何差别。当然，该函数现在能够调用支持透明功能的光栅化函数。使用透明功能时，速度会稍有降低。本可以增加一个多边形属性，用于指出纹理是否是透明的，然后，使用一条 switch case 语句来判断应调用哪个光栅化函数，但就现在而言这么麻烦不值得。另外，如果这样做，必须在建模程序中通过设置来指出附带的纹理是透明的，而各种有关光照和 alpha 混合的古怪设置已经足够多了。

有关软件就介绍到这里——余下的是数学计算。下面通过演示程序来实现每一种技术。

14.4.3　简单阴影

要实现简单阴影，只需使用 alpha 混合在投下阴影的物体下方绘制一个位图阴影即可。要实现这种效果，首先需要两样东西：阴影纹理和用作广告牌的多边形网格。先来看阴影纹理，它名为 shadow64_64.bmp，大小为 64×64 像素，其中离中心的距离不超过 64 像素的 RGB 值为（16, 16, 16），其他像素为（0, 0, 0）。图 14.11 是这个纹理图的负片版本（negative version），这样虽然看不到被视为透明的部分，但至少能够知道非透明部分的形状。



图 14.11　阴影纹理（反转的负片图像）

接下来，需要创建一个由两个三角形组成的正方形，然后将纹理映射到这个正方形上。另外，为让纹理的非透明部分看起来更像阴影，必须在正方形模型中启用 alpha 混合功能。读者可能还记得，本书前面使用 Caligari .COB 文件格式来支持这种特性。我们将模型的透明性设置为 filter 模式，然后设置透明度。在这个例子中，我们将使用这样的透明度：使阴影纹理看起来较暗，但又不太暗，以避免阴影非常黑。最后，应将广告牌模型的光照模型设置为固定着色，以免阴影反射光。

广告牌模型名为 shadow_poly_01.cob，位于附带光盘中本章对应的目录中。有了广告牌模型和阴影纹理，并设置广告牌模型的 alpha 等级，以便对阴影纹理中非透明部分进行 alpha 混合后，便可以编写演示程序了。

编写演示程序非常容易。为物体创建阴影的步骤如下：

1．使用 z 缓存渲染场景中所有的物体。

2．对于要为其创建阴影的每个物体，使用支持 alpha 混合和透明的纹理映射函数在物体正下方的地形上（如果有的话）渲染广告牌（第 2 遍渲染）。

进行两遍渲染的原因是，要对阴影与地形（或其下面的几何体）进行 alpha 混合，缓存中必须有用于 alpha 混合的图像。另外，由于假设阴影在地面上，因此不但需要确定地面的位置，还需要确定在离地面多高的地方绘制阴影。图 14.12 说明了这些需要考虑的因素。

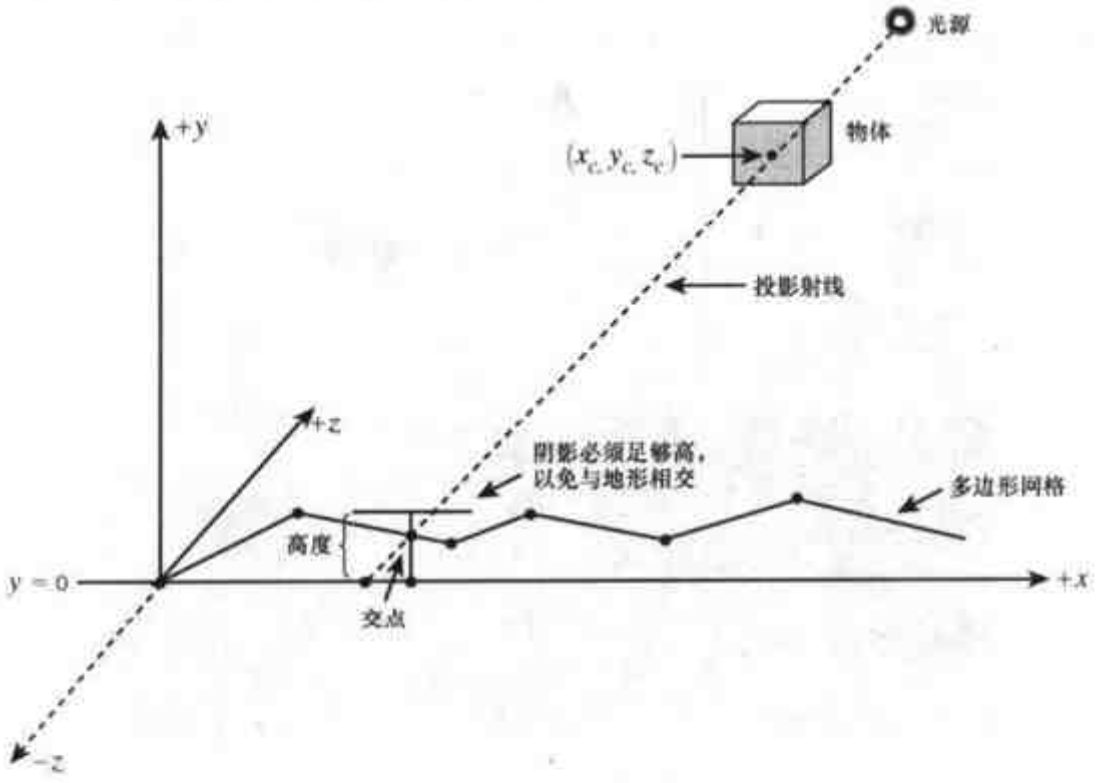


图 14.12　计算在多高的位置渲染阴影

在演示程序中，作者使用地形生成函数来创建了一个被冰覆盖的星球和一个绕该星球中心旋转的卫星。渲染过程如下：首先以常规方式渲染地形和卫星；然后渲染第 2 遍，但这次只渲染一个物体——广告牌。广告牌位于卫星的正下方，且与卫星居中对齐；同时根据地形来确定广告牌的高度，以免阴影让 z 缓存无效。当然，如果地形崎岖不平，效果将不会太好，因为无论怎样做，阴影都会与地形相交。

下面是演示程序中对阴影进行处理的代码片段：

```
// 重置渲染列表
Reset_RENDERLIST4DV2(&rend_list);

// 重置物体（仅当要进行背面消除和物体剔除时才需要这样做）
Reset_OBJECT4DV2(&shadow_obj);

// 计算阴影在哪个地形单元格上
cell_x = (obj_work->world_pos.x + TERRAIN_WIDTH/2) / obj_terrain.fvar1;
cell_y = (obj_work->world_pos.z + TERRAIN_HEIGHT/2) / obj_terrain.fvar1;

// 计算当前单元格的顶点索引
int v0 = cell_x + cell_y*obj_terrain.ivar2;
int v1 = v0 + 1;
int v2 = v1 + obj_terrain.ivar2;
int v3 = v0 + obj_terrain.ivar2;

// 计算 4 个顶点中最大的 y 坐标
terrain_height = MAX(  MAX(obj_terrain.vlist_trans[v0].y,
                 obj_terrain.vlist_trans[v1].y),
               MAX(obj_terrain.vlist_trans[v2].y,
                 obj_terrain.vlist_trans[v3].y) );

// 更新位置
shadow_obj.world_pos  = obj_work->world_pos;
shadow_obj.world_pos.y = terrain_height+10;

// 创建单位矩阵
MAT_IDENTITY_4X4(&mrot);

// 对物体的局部坐标进行变换
Transform_OBJECT4DV2(&shadow_obj, &mrot, TRANSFORM_LOCAL_TO_TRANS,1);

// 执行局部坐标到世界坐标变换
Model_To_World_OBJECT4DV2(&shadow_obj, TRANSFORM_TRANS_ONLY);

// 将物体插入到渲染列表中
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &shadow_obj,0);
```

在上述代码的后面，是设置渲染场境并进行渲染的代码。图 14.13 是演示程序运行时的屏幕截图。该

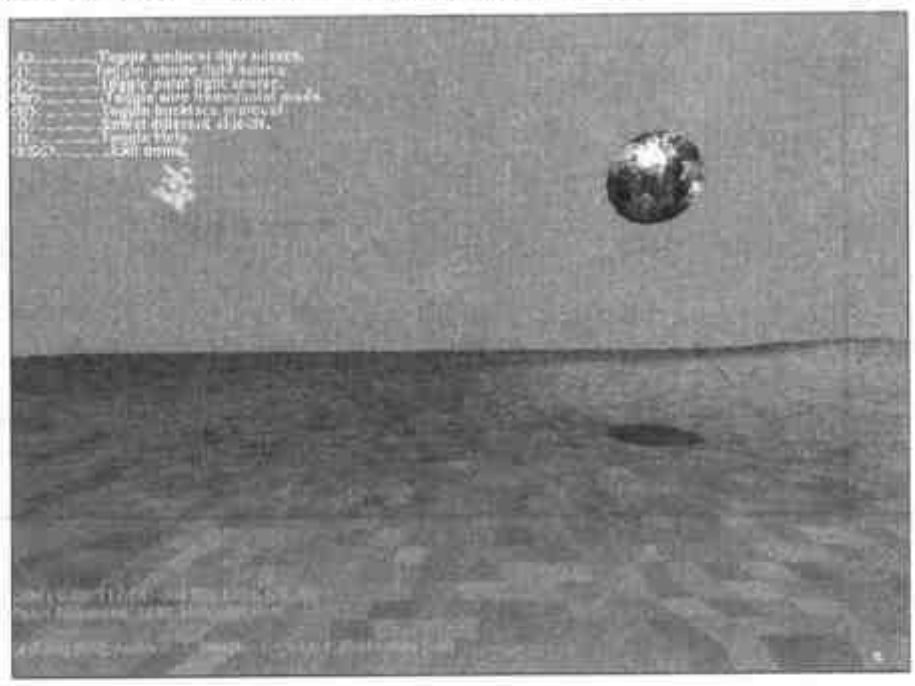


图 14.13　计算阴影演示程序的屏幕截图

程序名为 DEMOII14_1.CPP|EXE，要编译它，还需要 DirectX .LIB 文件和所有的库模块（T3DLIB1-12.CPP|H）。屏幕上有如何控制该演示程序的帮助信息，基本上，读者可以通过键盘在场景中移动、开/关光源以及在实心模式和线框模式之间切换。

14.4.4　缩放阴影

能够模拟阴影后，接下来更进一步，根据 3 个因素来缩放阴影：物体的大小、物体相对于地面的位置（高度）、导致阴影的光源的位置。

物体的大小是常数，可以不用考虑。以后可以使用一个缩放因子来将物体的大小考虑进来，但物体的高度和光源的高度/位置必须实时地考虑。必须推导出一种数学关系，其中考虑了光源和物体离地面的高度，如图 14.14 所示。

如图 14.14 所示，这里假设物体和光源都位于 y 轴上，物体在光源的下方。假设物体的高度为 h_0，半径为 R_0。当光源的高度为 h_1 时，阴影的半径为 R_1；当光源的高度为 h_2 时，阴影的半径为 r_2。假设光源的高度为 h_i 时，阴影的半径为 h_i；这样问题就变成了这样：R_0、R_i、h_0 和 h_i 之间存在什么样关系。

为此，可以利用三角形相似。由原点、h_i 和 r_i 构成的三角形与 h_i、h_0 和 R_0 构成的三角形相似，因此存在如下关系：

$$R_0/R_i = (h_i - h_0)/h_i$$

根据上述方程求解 r_i，结果如下：

$$R_i = R_0 * (h_i / (h_i - h_0))$$

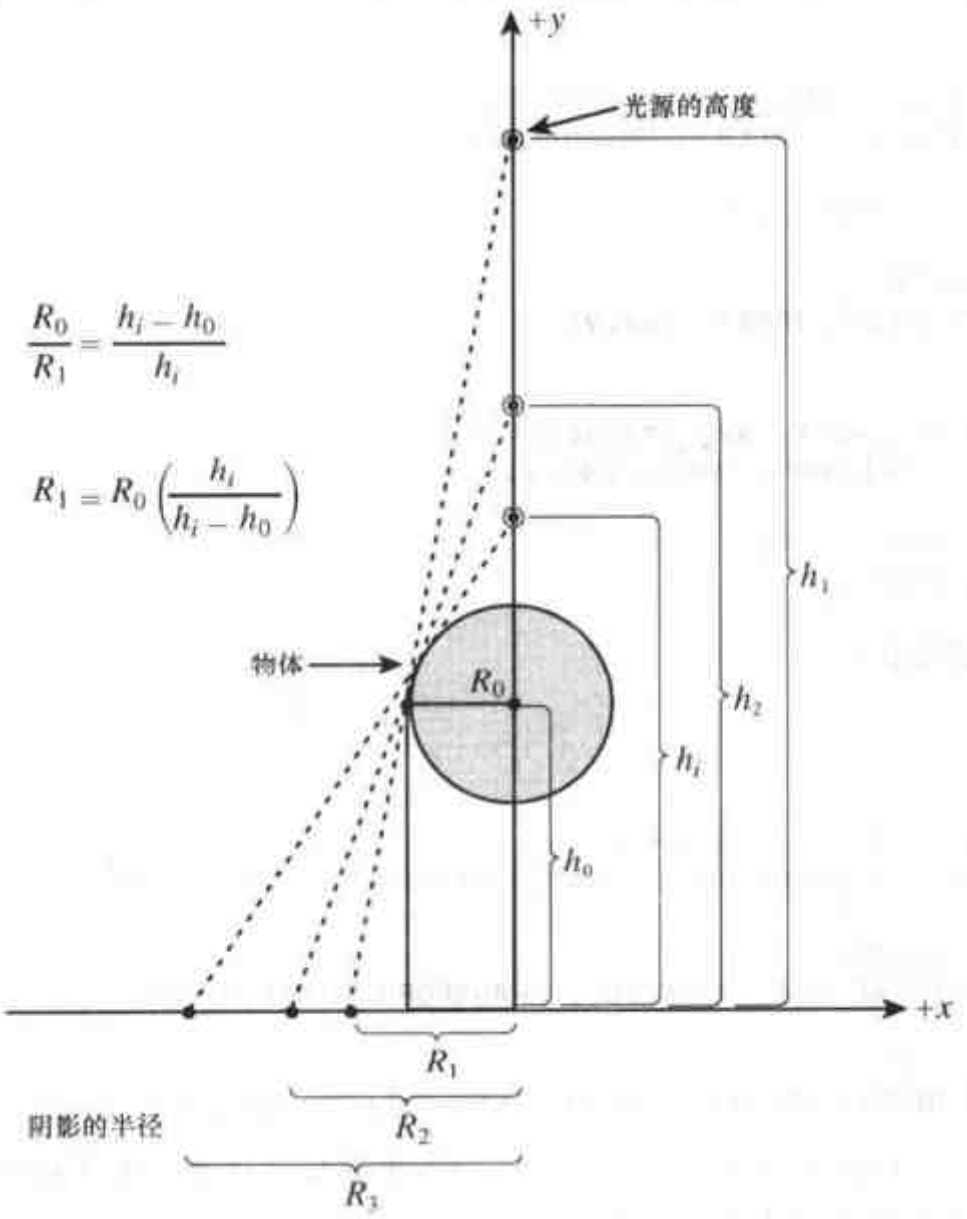


图 14.14　推导阴影缩放公式

如果光源像太阳一样，位于无穷远处，情况将如何呢？在这种情况下，光线将是平行的，因此阴影的半径为 R_0。

从上述公式可知，光源越高，阴影越小；光源越低，阴影越大。

接下来的演示程序将根据一个点光源的实际位置，使用上述公式来缩放阴影；然而，这里只考虑了光源（绿色点光源）的高度。

除了对阴影进行了缩放外，渲染阴影的方法与前面完全相同。下面是完成这项任务的代码片段：

```
// 进行第二遍渲染，绘制阴影

// 重置渲染列表
Reset_RENDERLIST4DV2(&rend_list);

// 重置物体（仅当要进行背面消除和物体剔除时才需要这样做）
Reset_OBJECT4DV2(&shadow_obj);

// 计算阴影在哪个地形单元格上
cell_x = (obj_work->world_pos.x + TERRAIN_WIDTH/2) / obj_terrain.fvar1;
cell_y = (obj_work->world_pos.z + TERRAIN_HEIGHT/2) / obj_terrain.fvar1;

// 计算当前单元格的顶点索引
int v0 = cell_x + cell_y*obj_terrain.ivar2;
int v1 = v0 + 1;
int v2 = v1 + obj_terrain.ivar2;
int v3 = v0 + obj_terrain.ivar2;

// 计算 4 个顶点中最大的 y 坐标
terrain_height = MAX(  MAX(obj_terrain.vlist_trans[v0].y,
                           obj_terrain.vlist_trans[v1].y),
                       MAX(obj_terrain.vlist_trans[v2].y,
                           obj_terrain.vlist_trans[v3].y) );

// 更新位置
shadow_obj.world_pos  = obj_work->world_pos;
shadow_obj.world_pos.y = terrain_height+25;

// 根据前面介绍的方法计算阴影的半径
// 将 hl 设置为绿色点光源的高度
hl = lights2[POINT_LIGHT_INDEX].pos.y;

// 计算阴影的半径
float rs = ks * ( obj_work->avg_radius[0] *
        (hl/ (hl - obj_work->world_pos.y)) );

// 生成一个单位矩阵，用于缩放变换
MAT_IDENTITY_4X4(&mrot);

// 设置 x-z 平面上的缩放比例
mrot.M00 = rs;
mrot.M11 = 1.0;
mrot.M22 = rs;

// 使用上述缩放矩阵对物体的局部坐标进行变换
Transform_OBJECT4DV2(&shadow_obj, &mrot, TRANSFORM_LOCAL_TO_TRANS,1);

// 执行局部坐标到世界坐标变换
Model_To_World_OBJECT4DV2(&shadow_obj, TRANSFORM_TRANS_ONLY);

// 将物体插入到渲染列表中
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &shadow_obj,0);
```

提示：在上述代码中，计算阴影半径时，乘以了一个常量 ks。这是一个虚构的因子，用于让阴影看起来更合理。有时候，正确并能保证看起来逼真。

新增的代码为粗体，它们只是实现了前面推导的数学公式。作者使用矩阵变换来缩放广告牌。读者可能还记得，对任何顶点进行缩放的矩阵类似于这样：

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

当然，在这个例子中，只需在 x-z 平面上进行缩放。

图 14.15 是演示程序 DEMOII14_2.CPP|EXE 运行时的屏幕截图。要编译该程序，还需要 T3DLIB1-12.CPP|H 以及 DirectX .LIB 文件。该程序的控制方法与前一个相同，但新增了按键 1 和 2 改变绿色点光源高度的控制方法。读者可尝试改变光源的高度，看阴影有何变化。另外，请按 W 键切换到线框模式，以查看为绘制阴影而渲染的网格——它是一个正方形。



图 14.15　支持阴影缩放的演示程序的屏幕截图

14.4.5　跟踪光源

现在，阴影看起来像阴影了，但它总是一个圆盘——如果物体为小型公共汽车，阴影仍为圆盘，不过这种问题很容易解决。真正的问题在于，阴影总是在投下阴影的物体的正下方，这是错误的。在前一个演示程序中，我们将绿色点光源用作产生阴影的光源，但只根据其高度来计算阴影的半径。现在，我们需要做的是，根据光源和物体的位置，计算阴影在 x-z 平面上的位置。

图 14.16 说明了我们要解决的问题。这里换换花样，使用参数化向量方程而不是三角形相似来解决这个问题。在图 14.16 中，点光源的位置为 P_L，物体中心的位置为 P_0，物体中心在地面上的投影点为 P_s。

在 3D 空间中，经过光源和物体中心与地面相交的参数化直线方程如下：

$$p_s = p_L + t * (p_0 - p_L)$$

用分量表示时，方程如下：

$$\begin{aligned} p_s.x &= p_L.x + t * (p_0.x - p_L.x) \\ p_s.y &= p_L.y + t * (p_0.y - p_L.y) \\ p_s.z &= p_L.z + t * (p_0.z - p_L.z) \end{aligned}$$

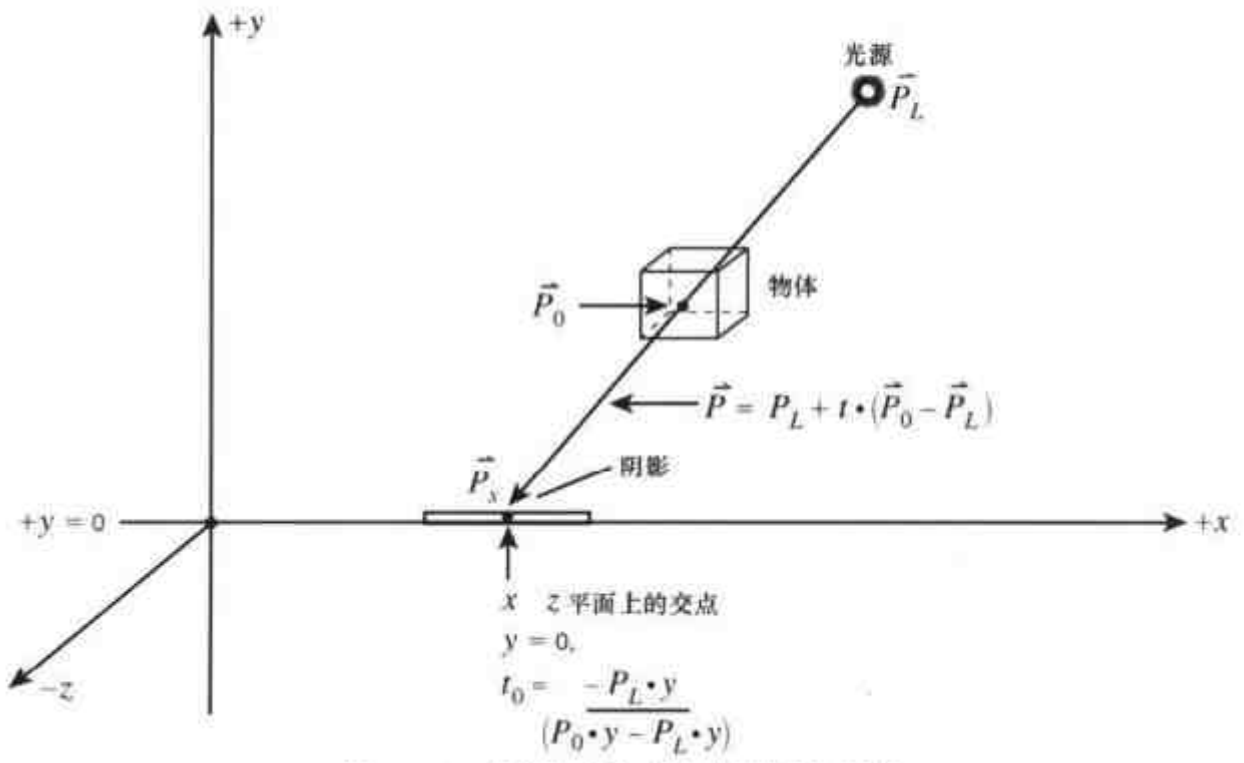


图 14.16　根据光源位置计算阴影的位置

我们感兴趣的是 y = 0 时，分量 x 和 z 的值。因此，可以令 P_s.y = 0，并求解 t 的值，进入将 t 代入其他两个方程，并求解分量 x 和 z 的值。

要求解 t 很容易：

$$p_s.y = p_1.y + t * (p_0.y - p_1.y) = 0$$

因此：

$$t = -p_L.y / (p_0.y - p_L.y)$$

确定 p_s 的坐标后，只需以 p_s 为中心渲染阴影便大功告成了。阴影将随光源的移动而移动，就像是由于物体遮掩而形成的。另外，还需加入缩放阴影的功能。

渲染流水线的实现与前一个演示程序相同：首先渲染地形和物体，然后根据前面推导出的数学公式确定阴影的位置，并使用 alpha 混合来渲染广告牌。该演示程序绘制的阴影相当逼真。

当然，还有两个问题需要解决：阴影的形状与物体毫无关系，总是为圆盘；没有对阴影进行错切。稍后将解决这两个问题。

来看一下生成阴影的新代码，它们根据光源和物体的位置，计算阴影在 y=0 平面上的位置。下面的代码摘自演示程序 DEMOII14_3.CPP：

```
// 重置渲染列表
Reset_RENDERLIST4DV2(&rend_list);

// 如果飞机座舱在阴影中，则使用第 0 帧中的图像
cockpit.curr_frame = 0;

int v0, v1, v2, v3; // 用于存储顶点索引

VECTOR4D pl, // 光源位置
     po, // 物体位置
     vlo, // 光源到物体的向量
     ps; // 阴影的位置
float  rs, // 阴影半径
     t;  // 参数 t

//////////////////////////////////////////////////////////////////////////////
// 绘制物体的阴影

// 重置物体（仅当要进行背面消除和物体剔除时才需要这样做）
Reset_OBJECT4DV2(&shadow_obj);

// 计算阴影在哪个地形单元格上
cell_x = (obj_work->world_pos.x + TERRAIN_WIDTH/2) / obj_terrain.fvar1;
cell_y = (obj_work->world_pos.z + TERRAIN_HEIGHT/2) / obj_terrain.fvar1;

// 计算当前单元格的顶点索引
v0 = cell_x + cell_y*obj_terrain.ivar2;
v1 = v0 + 1;
v2 = v1 + obj_terrain.ivar2;
v3 = v0 + obj_terrain.ivar2;

// 计算 4 个顶点中最大的 y 坐标
terrain_height = MAX(  MAX(obj_terrain.vlist_trans[v0].y,
          obj_terrain.vlist_trans[v1].y),
        MAX(obj_terrain.vlist_trans[v2].y,
          obj_terrain.vlist_trans[v3].y) );

// 更新位置
//shadow_obj.world_pos  = obj_work->world_pos;
shadow_obj.world_pos.y = terrain_height+25;

// 将光源 1 作为投影仪，计算投影的位置
// 使用局部变量使计算更容易理解

// 光源位置
pl = lights2[POINT_LIGHT_INDEX].pos;

// 物体位置
po = obj_work->world_pos;

// 创建从光源到物体的向量
```

```
VECTOR4D_Build(&pl, &po, &vlo);

// 计算投影线与平面 y=0 的交点
// 实际上，阴影离地面有一定的距离
t = -pl.y / vlo.y;

// 计算阴影中心的 x、z 坐标
shadow_obj.world_pos.x = pl.x + t*vlo.x;
shadow_obj.world_pos.z = pl.z + t*vlo.z;

// 接下来根据前面介绍的方法计算阴影的半径

// 光源高度
hl = lights2[POINT_LIGHT_INDEX].pos.y;
// 使用物体的平均半径来计算阴影半径
rs = ks * ( obj_work->avg_radius[0] * (hl/ (hl - obj_work->world_pos.y)) );

// 创建一个单位矩阵
MAT_IDENTITY_4X4(&mrot);

// 设置 x-z 平面上的缩放因子
mrot.M00 = rs;
mrot.M11 = 1.0;
mrot.M22 = rs;

// 将物体的局部坐标进行缩放
Transform_OBJECT4DV2(&shadow_obj, &mrot, TRANSFORM_LOCAL_TO_TRANS,1);

// 执行局部坐标到世界坐标变换
Model_To_World_OBJECT4DV2(&shadow_obj, TRANSFORM_TRANS_ONLY);

// 将物体插入到渲染列表中
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, &shadow_obj,0);
////////////////////////////////////////////////////////////////////////////
// 检测相机是否在阴影中
VECTOR4D vd;
VECTOR4D_Build(&cam.pos, &shadow_obj.world_pos, &vd);
float d = VECTOR4D_Length_Fast(&vd);

// 检测距离是否小于阴影半径的 1.5 倍
if (d < 1.5*rs)
  cockpit.curr_frame = 1; // 使用较暗的飞机座舱图像
else
  cockpit.curr_frame = 0; // 使用较亮的飞机座舱图像
```

计算阴影位置的代码几乎与前面介绍的方法完全相同；惟一新增的特性是：当相机位于阴影中时，更换飞机座舱的位图。如果读者玩过第一人称游戏，可能注意到了，当您进入较暗的区域时，玩家角色的光照强度将减弱，这是光照模型（或手工）降低玩家角色的环境光因子实现的，旨在模拟阴影效果。处于好玩的目的，在演示程序中实现了相同的功能。作者使用吉普车来表示飞机座舱，并提供了两个版本：一个较亮，一个较暗。如果相机位于阴影中，则使用较暗的图像；反之则使用较亮的图像，从而营造出飞机座舱位于阴影中的假象。

在实际的第一人称游戏中，可能使用 3D 几何体来表示飞机座舱、玩家、枪炮等，并让光照模型来进行着色。然而，光照模型并不知道阴影，因此您仍需要检测网格是否在阴影中，进而降低环境光项等，使网格变暗。在上述代码中，执行这种检测并交换位图的代码为粗体。

在该演示程序中，另一个新增的代码片段是，对于每个点光源（共有两个），都渲染物体的一个阴影，这在执行阴影计算时非常重要。对于每个光源，都需要计算每个物体的阴影；即使使用简单的阴影算法，计算量也将非常大。假设有 50 个物体和 3 个光源，将需要渲染 150 个阴影。虽然每个阴影只是一个带纹理的、由两个多边形组成的网格，但渲染它们时需要进行 alpha 混合，这将对性能带来一定的影响。

最后，为确保作者和读者对渲染算法的认识一致，请看图 14.17 所示的流程图。它说明了渲染流水线，包含如下步骤：

1. 渲染地形；
2. 渲染光源标记（marker）；
3. 渲染物体；
4. 光栅化图像；
5. 渲染阴影；
6. 使用 alpha 混合光栅化图像。

图 14.18 是演示程序 DEMOII14_3.CPP|EXE 运行时的屏幕截图。控制方法与前一个演示程序相同，但除了可以使用按键 1 和 2 来改变绿色点光源的高度外，还可以使用按键 3 和 4 来改变红色点光源的高度。但请注意，如果光源位置太低，将导致反转投影（inverted projection），出现怪异的现象。另外，请尝试将光源移到离地面非常高的位置，以模拟太阳，这样阴影将在卫星附近，大小与卫星相同。

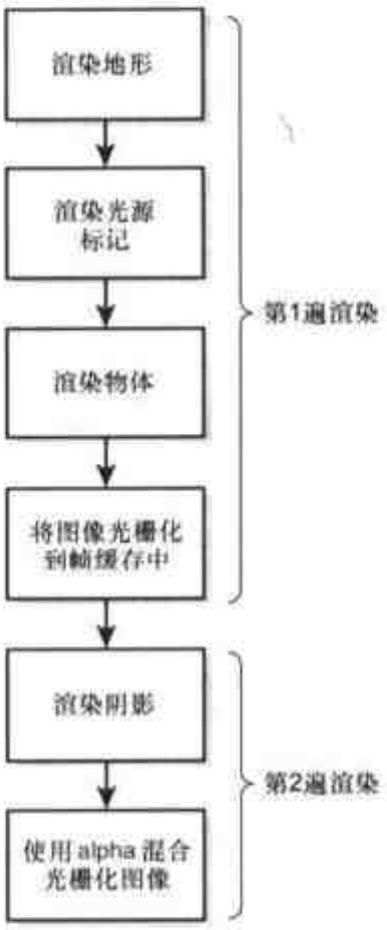


图 14.17 阴影渲染流程图



图 14.18 支持缩放和定位的阴影演示程序的屏幕截图

要编译该演示程序，需要库模块 T3DLIB1-12.CPP|H、主程序文件 DEMOII14_3.CPP|H 以及 DirectX .LIB 文件。

14.4.6 有关模拟阴影的最后思考

使用前面介绍的技术，可以实现一些非常棒的阴影效果。事实上，大多数硬件加速游戏采用的中庸之道都是使用定位、缩放和 alpha 混合来绘制阴影；从这种意义上说，我们做得已经很不错了。

读者可能想加入的最后一种特性是，将以光源为视点对 3D 物体进行投影得到的位图图像作为阴影。这样，阴影的形状将与物体相关。这样做惟一存在的问题是，如果物体旋转，阴影网格也必须跟着旋转。不过，如果旋转是绕 y 轴进行的（如图 14.19 所示），这种问题将很容易解决。

如果物体绕 y 轴旋转，只需将广告牌旋转同样的角度，并将物体的投影位图图像作为纹理。当然，如果物

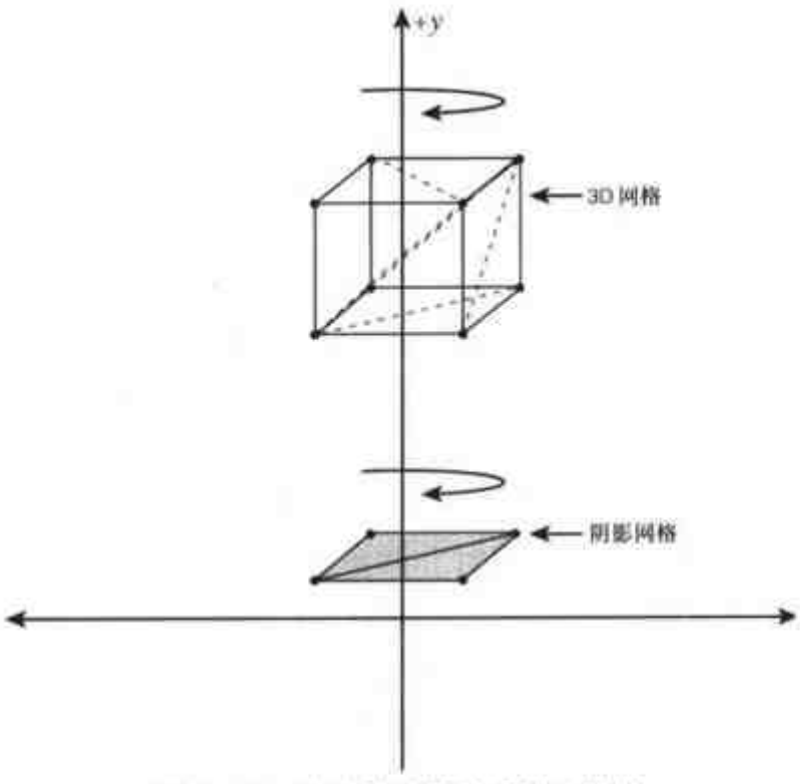


图 14.19 广告牌随物体旋转而旋转

体绕其他轴旋转，阴影看起来将是错误的；但是对很多游戏和物体来说，这种情况不会发生。加入这种功能很容易，只需 2～5 行代码；通过使用投影位图，可使阴影的真实感更强。

14.5　平面网格阴影映射

现在可以介绍一种更复杂的阴影技术了，它将物体投影到地面上，然后使用灰色和 alpha 混合绘制投影，使之看起来像阴影。图 14.20 说明了从光源出发，将物体网格上的顶点投影到地面上的情形。

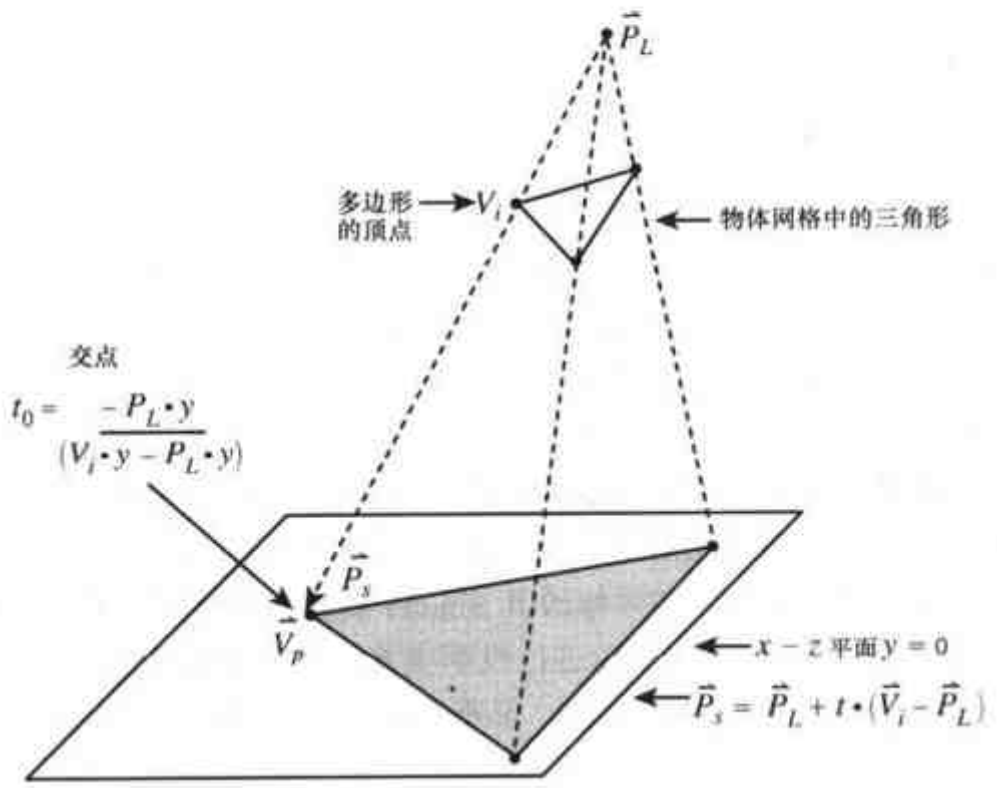


图 14.20　平面网格阴影映射

在前一个例子中，已经解决了这种问题；现在只需使之更为通用。我们要做的是，找出一种变换，将 3D 网格投影到一个平面上（这里为 x-z 平面），然后渲染投影得到的平面网格，并设置着色模式（禁用纹理映射和光照等），使之看起来像阴影。换句话说，我们将渲染物体在地面上的投影，将其作为阴影。

14.5.1　计算投影变换

我们需要的是这样一种变换：根据光源位置将物体网格中的顶点投影到地面上。下面来推导数学公式。

我们要对物体网格中的顶点 v_i 进行投影，光源的位置为 p_L，地面为 y=0 平面。假设地面上的投影点为 p_s，则计算公式如下：

$$p_s = p_L + t * (v_i - p_L)$$

用分量表示时，方程如下：

$$p_s.x = p_L.x + t * (v_i.x - p_L.x)$$
$$p_s.y = p_L.y + t * (v_i.y - p_L.y)$$
$$p_s.z = p_L.z + t * (v_i.z - p_L.z)$$

投影点在 x-z 平面上，因此 $p_s.y = 0$。据此，可以求解 t（将其称为 t_0），然后将其代入其他两个方程，求解出分量 x 和 z。

$$p_s.y = p_L.y + t * (v_i.y - p_L.y) = 0$$

因此：

$$t_0 = -p_L.y \ / \ (v_i.y - p_L.y)$$

将其代入其他两个方程，可得如下结果：

$$p_s.x = p_L.x + (-p_i.y \ / \ (v_i.y - p_i.y)) * (v_i.x - p_i.x)$$

$$p_s.y = 0$$
$$p_s.z = p_L.z + (-p_L.y / (v_i.y - p_L.y)) * (v_i.z - p_L.z)$$

根据用向量表示的线段方程可知：

$$\begin{aligned} p_s &= p_L + t_0 * (v_i - p_L) \\ &= p_L + t_0 * v_i - t_0 * p_L \\ &= t_0 * v_i + p_L * (1 - t_0) \end{aligned}$$

这种运算很适合用矩阵来表示，对应的矩阵如下：

$$M_{ps} = \begin{bmatrix} t & 0 & 0 & 0 \\ 0 & t & 0 & 0 \\ 0 & 0 & t & 0 \\ p_l.x*(1-t_0) & p_l.y*(1-t_0) & p_l.z*(1-t_0) & 1 \end{bmatrix}$$

将顶点 v_i 左乘上述矩阵时结果如下：

$$v_i * M_{ps} = [\ t_0*v_i.x + p_L.x*(1-t_0),\ t_0*v_i.y + p_L.y*(1-t_0),\ t_0*v_i.z + p_L.z*(1-t_0)]$$

这表明矩阵是正确的，但问题是，这个矩阵只适用于一个顶点。计算 t_0 的公式中包含 v_i，因此虽然矩阵是正确的，但只适用于 v_i。对于每个顶点 v_i，都需要重新计算 t_0。

看来使用矩阵进行变换并非好主意。最好的方法是，在循环中根据每个顶点的参数化方程求解 t_0，然后将其代入其他两个方程，以求解顶点变换后的 x 和 z 坐标。好消息是，由于这些变换和计算都是在世界空间中进行的，因此只需在计算阴影时进行局部坐标到世界坐标变换，无需决定阴影网格的位置。

假设已经根据物体和光源的位置通过投影计算得到了阴影网格，可以对其进行渲染了。但是，简单地渲染阴影网格并不能使其看起来像阴影，必须修改其 Shader 属性。为使阴影看起来像阴影，处理步骤如下：

1．将物体网格投影到地面上：对局部坐标进行投影变换，将结果作为变换后的坐标（别忘了，只要不覆盖模型的局部坐标，不管如何对模型进行处理，都不会破坏原来的模型。因此，总是可以对模型进行变换，将变换结果存储在变换后的顶点列表中）。

2．修改变换得到的阴影网格的光照和 Shader 属性，使其看起来像阴影。为此，只需保存每个多边形的颜色和属性，然后这样设置每个多边形：使用固定着色和 alpha 混合，将 alpha 值设置为大约 50%，不使用纹理。

3．将阴影网格传递给渲染流水线，然后恢复之前保存的颜色和属性。

基本上，我们使用物体本身来生成阴影：将物体网格投影在地面上，再修改网格中多边形的 Shader 属性，使之看起来像阴影，然后渲染它，并恢复原来的属性。

下面是演示程序 DEMOII14_4.CPP 中完成这项工作的代码：

```
// 重置渲染列表
Reset_RENDERLIST4DV2(&rend_list);

//////////////////////////////////////////////////////////////////////////
// 将物体的顶点投影到地面上来得到物体的阴影

// 重置物体（仅当要进行背面消除和物体剔除时才需要这样做）
Reset_OBJECT4DV2(obj_work);

int pcolor[OBJECT4DV2_MAX_POLYS],  // 用于保存颜色
  pattr[OBJECT4DV2_MAX_POLYS];   // 用于保存属性

// 保存每个多边形的属性和颜色
for (int pindex = 0; pindex < obj_work->num_polys; pindex++)
  {
  // 保存属性和颜色
  pattr[pindex] = obj_work->plist[pindex].attr;
  pcolor[pindex] = obj_work->plist[pindex].color;

  // 重新设置属性以渲染阴影
  obj_work->plist[pindex].attr  = POLY4DV2_ATTR_RGB16 |
                  POLY4DV2_ATTR_SHADE_MODE_CONSTANT |
                  POLY4DV2_ATTR_TRANSPARENT;
```

```
  obj_work->plist[pindex].color  = RGB16Bit(0,0,0) + (2 << 24);

  } // end for pindex

// 创建一个单位矩阵
MAT_IDENTITY_4X4(&mrot);

pl = lights2[POINT_LIGHT_INDEX].pos;

// 对物体的局部/模型顶点坐标进行变换，将结果存储到变换后的顶点列表中
for (int vertex=0; vertex < obj_work->num_vertices; vertex++)
  {
  POINT4D presult; // 用于存储变换结果

  VECTOR4D vi;

  VECTOR4D_Add(&obj_work->vlist_local[vertex].v,&obj_work->world_pos, &vi);

  // 计算 t0
  float t0 = -pl.y / (vi.y - pl.y);
  // 对顶点进行变换
  obj_work->vlist_trans[vertex].v.x = pl.x + t0*(vi.x - pl.x);
  obj_work->vlist_trans[vertex].v.y = 25.0; // pl.y + t0*(vi.y - pl.y);
  obj_work->vlist_trans[vertex].v.z = pl.z + t0*(vi.z - pl.z);
  obj_work->vlist_trans[vertex].v.w = 1.0;

  } // end for index

// 将物体插入到渲染列表中
Insert_OBJECT4DV2_RENDERLIST4DV2(&rend_list, obj_work,0);
```

接下来根据下一个光源来创建和渲染物体的阴影网格，直到考虑了所有的光源为止（在这个例子中，只有两个光源）。然后，恢复网格中多边形的属性和颜色，使网格恢复正常：

```
// 恢复属性和颜色
for (pindex = 0; pindex < obj_work->num_polys; pindex++)
  {
  // 恢复属性和颜色
  obj_work->plist[pindex].attr = pattr[pindex];
  obj_work->plist[pindex].color = pcolor[pindex];
  } // end for pindex
```

渲染顺序与以前相同：首先渲染地形，然后是光源标记和物体本身；接下来进行第二遍渲染——渲染阴影，因为需要将其同背景图像进行 alpha 混合。

这种阴影技术的效果非常好，图 14.21a 和 14.21b 分别是线框模式和实心模式下，这种阴影技术的演示程序的屏幕截图。该演示程序名为 DEMOII14_4.CPP|EXE，要编译它，还需要库模块 T3DLIB1-12.CPP|H 以及 DirectX .LIB 文件。控制方法与前一个演示程序完全相同，强烈建议读者使用 W 键切换到线框模式，以查看使得阴影从几何学上说是正确的错切（skewing）。

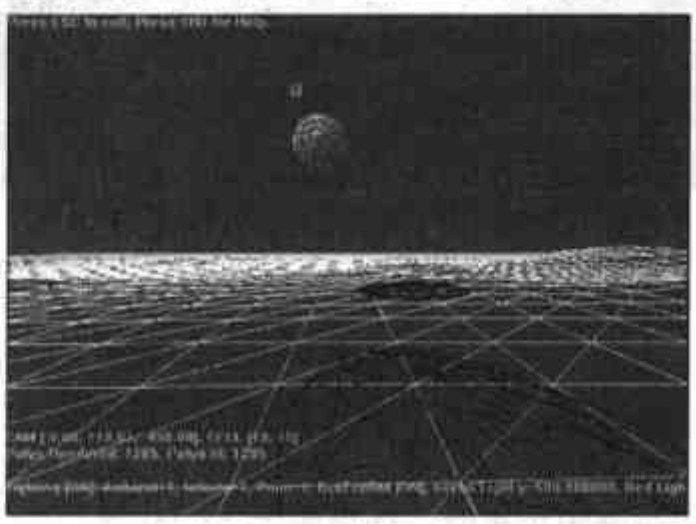


图 14.21a　线框模式下平面阴影演示程序的屏幕截图



图 14.21b　实心模式下平面阴影演示程序的屏幕截图

14.5.2 优化平面阴影

现在来说说坏消息：我们渲染了大量不需要渲染的多边形。我们将一个立体模型压缩为二维的，因此可能渲染背面或重叠的几何体。我们要做的是，只对物体轮廓进行投影，以得到阴影网格。

图 14.22 是一个简单立方体及其轮廓。窍门是找出轮廓，这样的方法有很多。轮廓由这样的边组成：共用它的两个多边形相对于光源而言分别是正面和背面。

在图 14.22 中，$\mathbf{e}_i$ 是一条轮廓边。它是多边形 $\mathbf{p}_0$ 和 $\mathbf{p}_1$ 共用的边，相对于光源而言，多边形 $\mathbf{p}_0$ 为正面，而 $\mathbf{p}_1$ 为背面。因此，我们可以使用这种算法来找出轮廓边，进而得到轮廓。问题是，这种方法只适用于没有孔且其中的几何体相互贯通的物体。这也没什么，但归根结底，这样做值得吗？

也许是的。作者发现，轮廓边方法更适合用于各种基于模板缓存或体阴影（volume shadow）的算法。问题是，在很多情况下，为节省一些时间而花费的时间比直接绘制多边形还长！另一种技巧是，生成阴影时，对每个物体的简化网格进行投影。简化网格中的多边形数量为完整模型的 10%，甚至更少，因此为绘制阴影，性能最多降低 10%——这不赖。

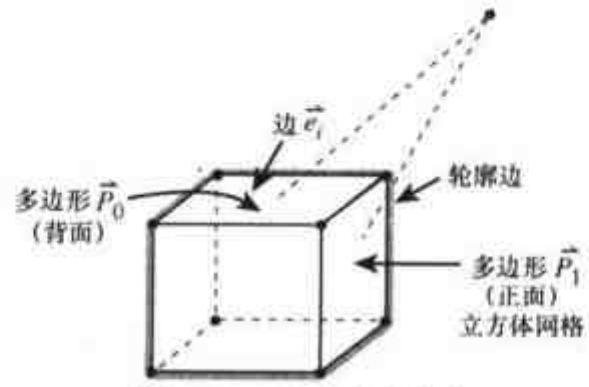


图 14.22 立方体的轮廓

14.6 光照映射和面缓存技术简介

这个主题介绍起来有些难，介绍它之前，请读者先来看一个没有使用光照映射的 3D 场景，如图 14.23a 所示。

接下来请看图 14.23b，发现了差别吗？真是有天壤之别！读者可能觉得这没有什么了不起的，使用我们的 Gouraud Shader 和其他光照系统也能得到类似的效果。事实上，并非如此，原因有两个。首先 Gouraud 着色是基于顶点的，无法绘制详细的细节；其次，我们无法在墙面、物体上创建阴影其自身的阴影等。这是光照映射发挥作用的领域。



图 14.23a 一个没有使用光照映射的 Quake 关卡



图 14.23b 一个使用光照映射的 Quake 关卡

光照映射是一种古老的技术，其最初的应用可追溯到 8 位图像时代，甚至更早，它基本上是在纹理上映射纹理。其中的技巧很简单：不是实时地执行光照计算，而是通过离线光照计算来计算场景中的光照强度、阴影等。这种工作也可以由艺术师手工绘制出光照效果来完成。通过计算或手工绘制得到的 2D 图像被称为光照图（light map）。

例如，将图 14.24a 所示的纹理和图 14.24b 所示的光照纹理图相乘时，结果如图 14.24c 所示。本书前面介绍光照模型时，做过类似的计算。然而，我们现在要做的是，大范围地应用这种技术，将范围扩大到

整个房间和网格。

光照映射的基本思想是，使用光照纹理图来调制被映射到世界几何体和游戏中物体上的纹理。

读者的第一反应可能是赞不绝口，但接下来可能会问，为何不创建预先经过光照处理的纹理呢？可以这样做，但如果有一个大型环境，环境中有 1000 个墙面/表面，每个墙面都有一个 256×256 的纹理，情况将如何呢？如果初始纹理为如图 14.25 所示的图像，将需要 1000 个这样的拷贝，并预先对其进行光照处理。这无疑是浪费内存。读者可能会反驳，如果使用同一个纹理，仍需要 1000 个光照图！确实如此，但情况并不完全相同。关键之处在于，光照图的分辨率无需与纹理相同。例如，对于 256×256 的纹理，使用 32×32 的光照图时，几乎看不出有任何影响。



图 14.24　单个纹理的光照映射



图 14.25　墙面纹理

图 14.26 是一个使用低分辨率的光照图来调制纹理的例子，其中包含原始纹理、32×32 的光照图和调制后的纹理。是的，看起来有些粗糙，但如果调制前使用滤波器对光照图进行处理，并将结果保存在缓存中，将得到一幅非常完美的图像。也就是说，以分辨率比目标纹理低很多倍的方式存储光照图，然后使用平均滤波器将其解压缩到缓存中，再使用处理后的光照图（假设为 256×256）对纹理（也为 256×256）进行调制。在最糟糕的情况下，每个光照图只占用 32×32 = 1024（单色）字节的内存，而不是 65536 字节内存——少了 64 倍。



图 14.26　使用低分辨率的光照图

然而，由于进行解压缩和平均，结果看起来相同。当然，读者可能知道，解压缩只不过是对光照图进行插值而已。重要的是，光线从本质上说是扩散和模糊的，柔和的阴影比粗糙的边缘看起来更美观。使用低分辨率的光照图时，在调制之前使用平均滤波器来减少光照图中的锯齿可达到这种目的。

光照映射的整个处理过程如图 14.27 所示。首先对光照图进行解压缩，得到分辨率更高的光照图，并将其复制到缓存中（当然，这一步是可选的）。然后使用光照图对纹理进行调制，这意味着将光照图中的像素值与纹理图中的像素值相乘，将结果存储到缓存中，并将其用作最终映射到多边形上的纹理。

这就是光照映射的全部内容。使用一个 2D 光照图对一个 2D 纹理进行调制，然后将调制结果作为纹理映射到几何体上。这种技术很不错，因为其开销只是根据原始纹理生成光照映射纹理的计算量。计算量可能相当可观，对于 256×256 的纹理，需要对 64 000 个像素进行计算，每个像素需要执行 1～3 次乘法运算（取决于是单色还是彩色），还有加法运算以及比较运算（截取）。假设每个像素的计算时间为常数 C_m，则对于 m×n 的纹理和 m×n 的光照图，总计算时间至少为 C_m*m*n。这是很大的计算量，每次对纹理进行光照映射并进行渲染后，我们都将结果丢弃，这是面缓存技术（surface caching）发挥作用的舞台。

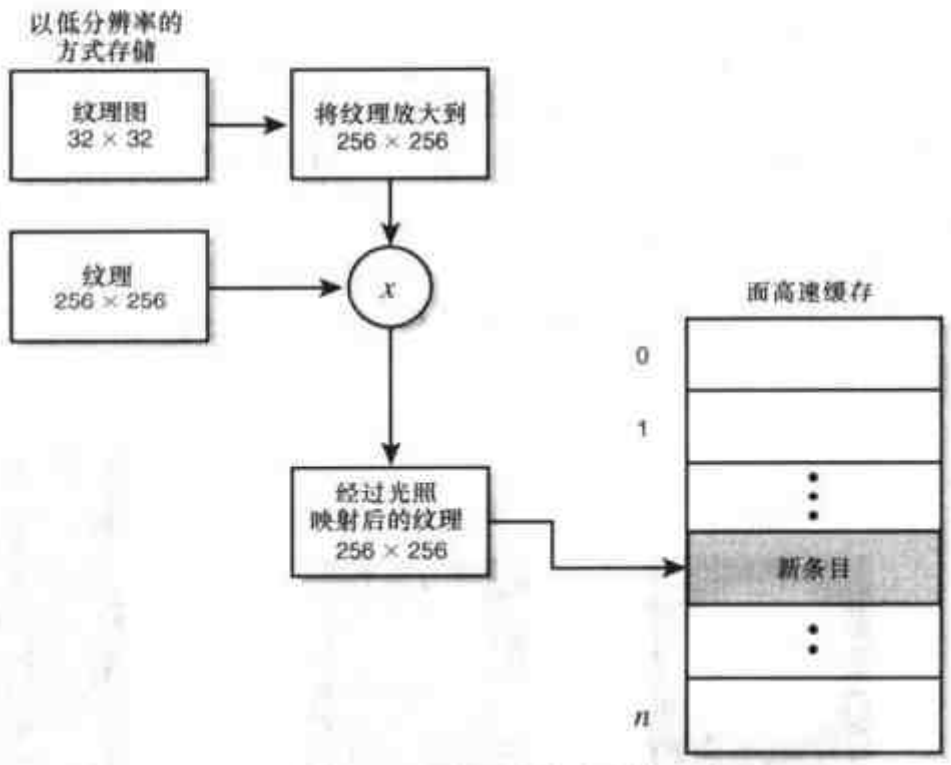


图 14.27　光照映射流水线

14.6.1　面缓存技术

面缓存技术指的是将光照映射计算的结果保存起来。例如，假设在一个房间中，有 10 个纹理需要进行光照映射。如果玩家暂时不会离开这个房间，为何不计算这些纹理的光照纹理映射版本，并将其保存到高速缓存（cache）中供下一帧使用呢？这样，如果高速缓存有所需的光照映射纹理，则无需重新计算。您不但不丢弃光照映射计算结果，还更新计算结果：有纹理需要进行光照映射时，执行相应的计算；光照映射纹理变得陈旧而不再需要时，将其删除。在光照映射中，面缓存技术的工作原理如下：

```
初始化纹理高速缓存，删除其中所有的纹理
Begin Scene
For(场景中的每个纹理 T)
if(面高速缓存中有纹理 T) then 使用它来渲染多边形
else
   Begin
   找到用于纹理 T 的光照图，使用它来调制纹理 T，并将结果存储到高速缓存中
   如果高速缓存已满，删除最近用得最少的纹理
   End else
End Scene
Repeat
```

非常简单。基本上，只是将每次光照映射的计算结果存储到高速缓存中，每当有相同的需求时使用高速缓存中的结果。这样，根据光照图计算出调制后的纹理后，便可以实现漂亮的纹理映射和阴影，而不需要任何开销。这难以置信，但确实如此。

当然，任何如此好的技术都存在问题，例如：

- 光照图应该多大？
- 基于多边形还是面（一组共面的多边形）来指定光照图？
- 应对光照图进行压缩吗？
- 光照图应为单色还是彩色（以实现彩色效果和阴影）？
- 面高速缓存应多大？

这个清单可以不断增长下去。然而，这些都需要经验才能做出的决策，且答案随具体情况而异。就现在而言，我们的目标是实现这种技术，以说明它是如此的简单，而效果又令人惊讶。

14.6.2　生成光照图

前面指出过，可以使用算法来生成光照图，也可以由艺术师手工绘制。如果您决定使用算法来生成光

照图，很可能是为了光照映射来显示美妙的阴影、黑暗的裂缝等。为实现这种效果，惟一的途径是使用离线光照系统，它使用光线跟踪或执行其他光照算法，如光子映射（photon mapping）或辐射度方法（radiosity）。

光线跟踪实现起来并不难，但效果不如辐射度方法，后者实际上是计算场景中的能通量（energy flow）。图 14.28 是一幅使用辐射度方法进行光照计算得到的图像，它看起来就像是真的。但对我们来说，所需的只是光照图的光照计算部分。这里不打算介绍辐射度方法，因为它还不是实时的；介绍辐射度方法、光子映射和光线跟踪的优秀图书有很多。

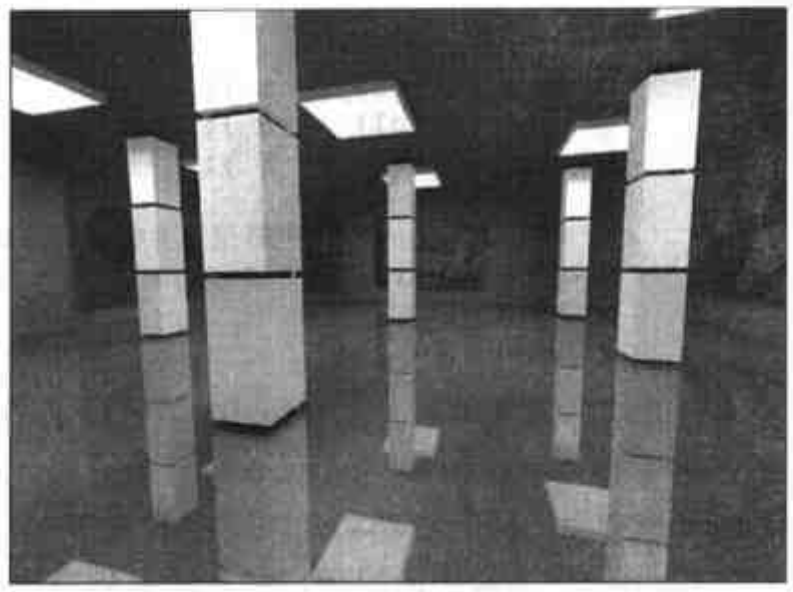


图 14.28　使用辐射度方法进行光照计算得到的场景

第二种选择也需要使用上述技术：在诸如 Pov-Ray、trueSpace 或 3D Studio Max 等 3D 建模工具中加载几何体，然后使用这些工具来创建一个经过光照处理的单色场景，再通过改变相机的位置，将其切割成光照图。

最后，也可以让艺术师绘制光照图，通过反复尝试，最后得到栩栩如生的光照图。采用哪种方式完全取决于您，在大多数情况下，作者结合使用这三种方式。例如，id Software 使用自己创建的工具来计算光照图，但众多其他的公司使用现成的工具，如 Pov-Ray 或 3D Studio Max，并编写一个接口或插件来利用其中的光线跟踪或辐射度系统，而不必浪费时间自己编写这样的工具。如果读者只是想熟悉光照映射技术的用法，建议手工绘制映射图，作者编写演示程序时就是这样做的。

14.6.3　实现光照映射函数

其实，我们已经编写了光照映射代码，光照映射只不过是使用一个纹理来调制另一个纹理而已，DEMOII8_4.CPP|EXE 正是这样做的，只是对大型纹理来说计算量很大。作者以该演示程序中的代码为基础，对其进行了优化，加入了使用光照图对纹理进行调制的代码。这样，速度非常快，效果也很不错。然而，像这样的算法亟需使用 SIMD（单指令多数据）来优化，这是奔腾 III 以上处理器的一种特性。也许我们会在本书最后一章进行这种优化，但现在暂时不去管它。下面是对纹理进行光照映射的代码段：

```
// 执行光照映射 ////////////////////////////////////////////

// 使用光照图对将被映射到目标多边形上的纹理进行调制
// 将结果用作渲染时使用的纹理

USHORT *sbuffer = (USHORT *)texture_copy.buffer;
USHORT *lbuffer = (USHORT *)lightmaps[curr_lightmap].buffer;
USHORT *dbuffer = (USHORT *)obj_terrain.texture->buffer;


// 对纹理进行调制
for (int iy = 0; iy < texture_copy.height; iy++)
  for (int ix = 0; ix < texture_copy.width; ix++)
      {
      int rs,gs,bs;  // 用于存储纹理中像素的 R、G、B 值
      int rl, gl, bl; //用于存储光照图中像素的 R、G、B 值
      int rf,gf,bf;  // 用于存储调制结果

      // 从纹理中提取像素
      USHORT spixel = sbuffer[iy*texture_copy.width + ix];

      // 提取像素的 R、G、B 值
      _RGB565FROM16BIT(spixel, &rs,&gs,&bs);

      // 从光照图中提取像素
      USHORT lpixel = lbuffer[iy*texture_copy.width + ix];

      // 提取像素的 R、G、B 值
```

```
_RGB565FROM16BIT(lpixel, &rl,&gl,&bl);

// 计算调制结果
rf = ( (rs*rl) >> 5 );
gf = ( (gs*gl) >> 6 );
bf = ( (bs*bl) >> 5 );

// 将结果重组成一个 RGB 值，并将其写入到缓存中
dbuffer[iy*texture_copy.width + ix] = _RGB16BIT565(rf,gf,bf);

} // end for ix
```

该算法非常简单，它使用 lbuffer 中的光照图对 sbuffer 中的纹理进行调制，并将结果存储到 dbuffer 中。计算是在 RGB 空间中对每个像素进行的，因此需要使用两个嵌套的循环，一个用于 x，一个用于 y。循环遍历纹理空间，提取纹理和光照图中的 RGB 值，将它们相乘，然后将结果存储到目标缓存 dbuffer 中，最后将 dbuffer 作为最终的纹理，将其映射到多边形网格上。

介绍使用光照映射的演示程序之前，先来看一个没有使用光照映射的演示程序，如图 14.29 所示。

这是演示程序 DEMOII14_5.CPP|EXE 的屏幕截图，该演示程序将被用于同光照映射演示程序进行比较。基本上，除了使用吊扇作为生成阴影的物体外，它与演示程序 DEMOII14_4.CPP|EXE 相同。吊扇模型如图 14.30 所示。

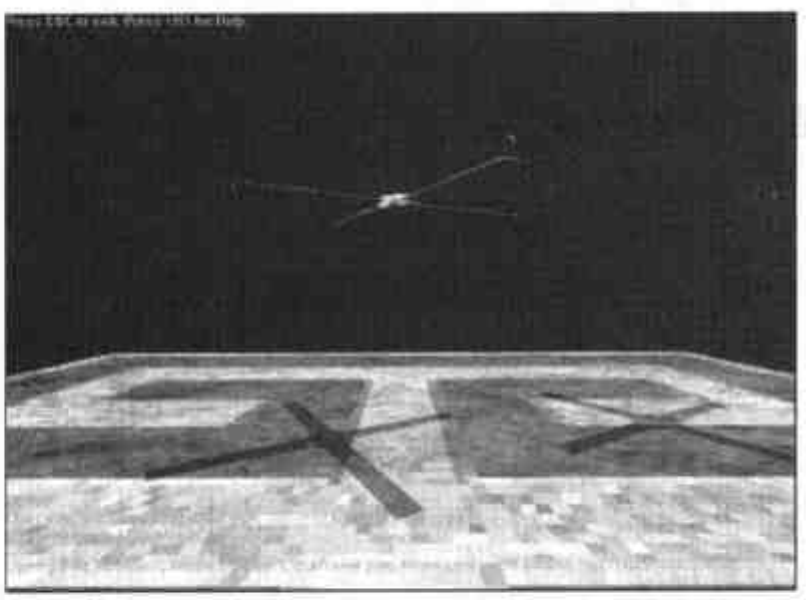


图 14.29　包含一台吊扇的平面阴影演示程序的屏幕截图

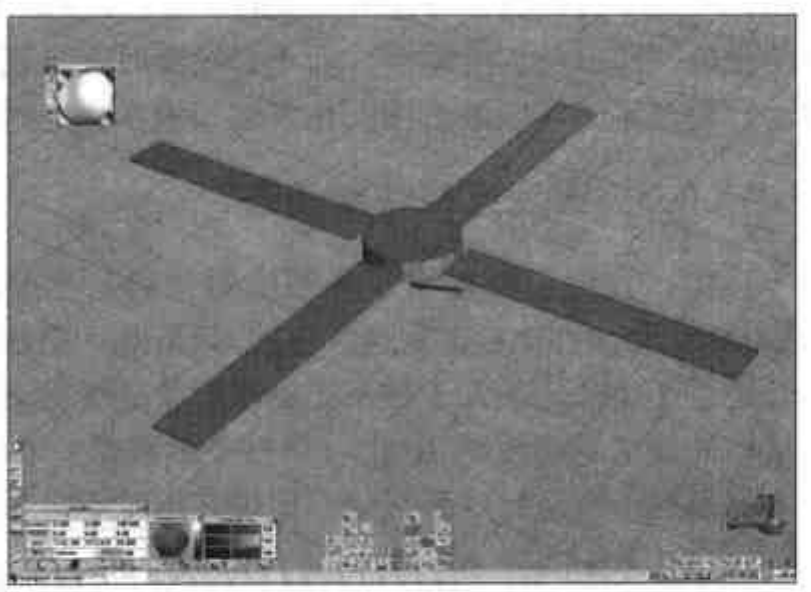


图 14.30　吊扇模型

有了可供比较的演示程序后，来看一看光照映射演示程序，它名为 DEMOII14_6.CPP|EXE，新增了几项内容。首先需要创建吊扇模型，为此，使用的是 trueSpace，花了 5～10 分钟的时间。它由一个圆柱体和 4 块叶片构成，文件名为 fan_01b.cob。其次，需要一个映射到地面（一个动态生成的地形）的纹理，如图 14.31 所示。



图 14.31　演示程序中使用的地面纹理

最后，需要光照图。这是惟一一项比较棘手的内容。作者使用建模程序 trueSpace 创建了吊扇的俯视图，将其大小设置为 256×256，并在 16 位颜色模式对其进行渲染，如图 14.32 所示。

得到 2D 光照图后，在 Paint Shop Pro 中加载它，然后以每次 10 度的方式对其进行旋转，创建一系列的光照图，如图 14.33 所示。有了模型和光照图后，创建演示程序的工作非常简单。

DEMOII14_6.CPP 是一个非常简单的循环：选择一个光照图，使用它来调制地形纹理，然后进入渲染阶段。在渲染阶段中，首先使用经过光照映射后的纹理来渲染地形，然后渲染吊扇的 3D 模型，整个渲染工作是一遍完成的，因为没有使用 alpha 混合。实际的光照映射工作是在纹理调制阶段完成的。光照映射的效果非常好，如图 14.34 所示。要编译该演示程序，除了主文件 DEMOII14_6.CPP 外，还需要 T3DLIB1-12.CPP|H，另外，别忘了链接 DirectX .LIB 文件。

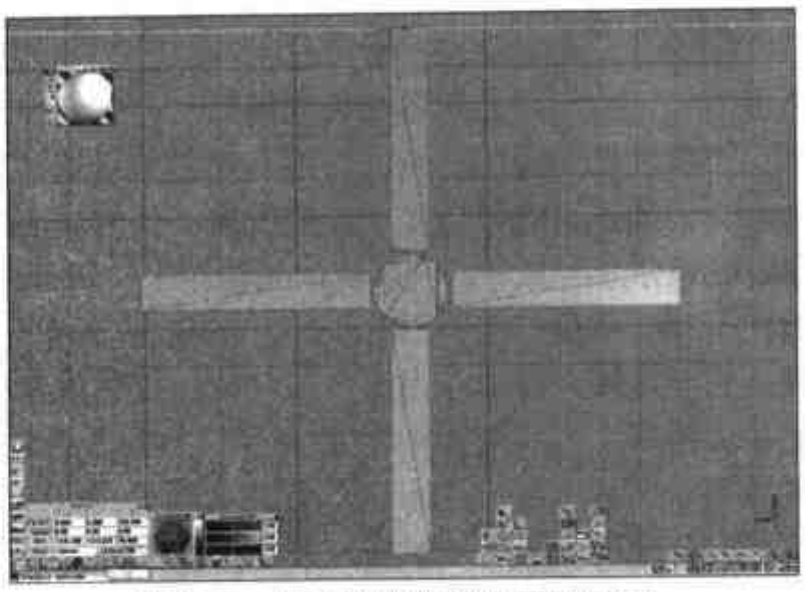


图 14.32　用于绘制吊扇阴影的光照图

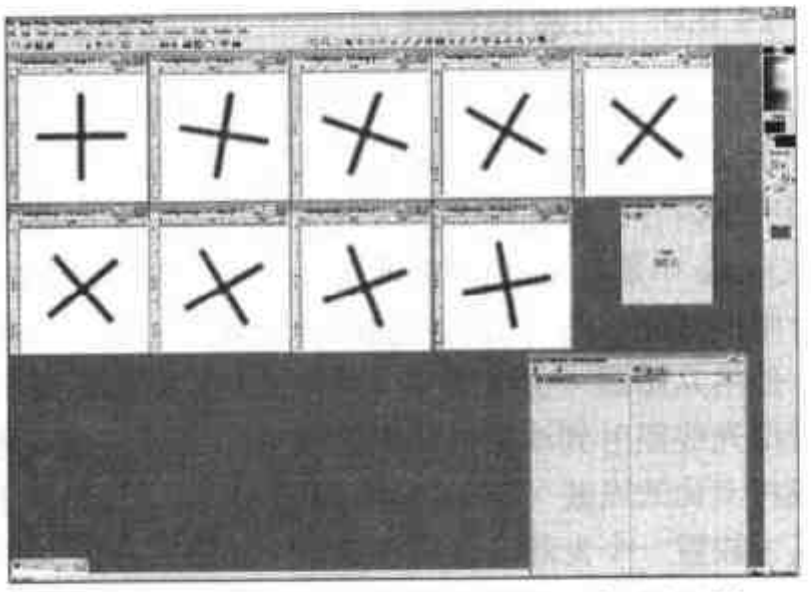


图 14.33　通过旋转和图像处理得到的一系列光照图



图 14.34　光照映射演示程序运行时的屏幕截图

14.6.4　暗映射（dark mapping）

没有人规定只能绘制阴影——也可以绘制光斑（light）。事实上，光照映射真是用词不当，因为其效果是加暗而不是增亮——调制算法最多只能让像素的亮度保持不变，大多数情况下会使像素变暗。

为创建一个暗映射演示程序，作者对 DEMOII14_6.CPP|EXE 进行修改，使之加载反向极性（opposite polarity）光照图。也就是说，对前面的 9 个光照图进行反转（invert）处理，如图 14.35 所示；然后进行模糊化处理，使光亮部分更突出。修改后的演示程序名为 DEMOII14_6.CPP|EXE，图 14.36 是它运行时的屏幕截图。如果读者运行这两个演示程序将发现，光源对光照效果的影响不大，这是因为光照效果是使用光照映射实现的。

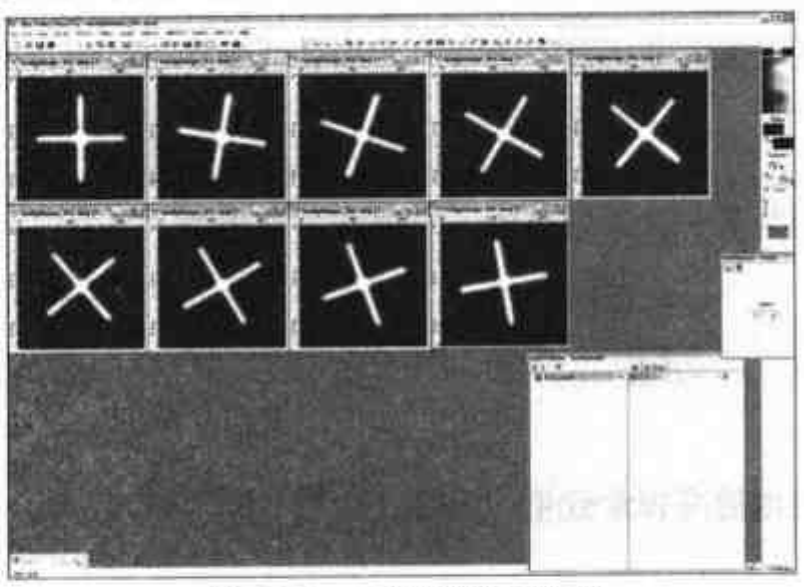


图 14.35　反转后的暗映射图



图 14.36　暗映射演示程序运行时的屏幕截图

14.6.5 光照图特效

在本章结束前，介绍一些使用光照图能够实现的特殊效果。首先，光照图的最大问题是，它们在很大程度上说是静态的，不过这没什么大不了的——没有规定不能创建光照图动画。当然，您必须认识到这会占用大量内存，但在不多的区域中使用光照图动画很容易。例如，可以使用多个光照图来实现闪烁的光源（像 Quake 中那样）、色彩效应等。另外，假设一个大房间被划分为四部分，可以使用光照图来实现开/关一部分中的灯光，而不影响其他部分的效果，条件是光照图不相互重叠。

使用光照图可实现的另一种特效是复杂的阴影动画，它非常棒。例如，当房门打开时，可使用光照图来模拟光线照射到漆黑房间中的效果，当光照强度达到最大值后，再启用正常光照。还可以使用光照图来实现将变化的光线（animation of light）投影到物体上的效果。例如，可以在圆柱体上裹上透明的塑料，并在其中放置一个光源（这是在模型空间中完成的）；然后在 3D 空间中旋转该模型，并将纹理本身用作光照图（这实际上被称为透视纹理，但理念是相同的，也是基于光照映射的）。

最后，一个目前还与之关系不大的领域是程序性（procedural）光照图。通过使用数学公式来生成纹理，然后将其用作光照图，可以实现云彩、烟雾和其他不断变化的效果，这也以某种方式对场景中的光照效果进行了调制。

14.6.6 优化光照映射代码

从算法上说，光照映射代码已经非常好了，但总是有办法可以进一步提高其速度。首先，可以使用查找表来代替乘法运算。然而，当前高速缓存一致性可能降低速度，而乘法运算只需一个时钟周期，因此作者认为这样做并不会有多大帮助。另一种优化方法是，使用 SIMD 指令，这样可以同时执行 4 次光照映射计算。这是绝对可行的，应优先考虑。

暂时抛开调制代码不谈，真正的问题是，如何确保将光照映射功能加入引擎中时，不会使速度降低很多。前面讨论了面高速缓存技术，它用于只生成光照映射纹理一次，然后重用它们，这是最好的优化方法。另外，还应使用较小的纹理图（如 32×32，或分辨率为纹理图的 1/16～1/32），以便高速缓存能够容纳所有的光照图，这样纹理调制阶段的运行速度将非常快。

总之，在作者看来，一个使用面高速缓存和低分辨率光照图（支持单色和彩色），同时支持光照图动画的简单系统，是一个非常强大的系统。

14.7 整 理 思 路

现在，读者能够实时地绘制阴影和进行光照映射了，我敢断定，读者一定会对此着迷。然而，别忘了，在每一帧中可用的 CPU 周期有限，虽然根据每个光源绘制物体的多边形透视阴影的效果非常好，但对于很多物体来说，使用广告牌来绘制阴影可达到同样的效果，尤其是当物体离视点很远时。另外，在使用阴影还是光照映射方面拿不定主意时，应使用光照映射，尤其是对室内场景而言。在软件或硬件引擎中，无论如何进行复杂光照处理，效果都不可与光照映射相媲美。

最后，尝试进行两遍渲染：先从一个角度渲染场景，在从另一个角度渲染场景。可能是先渲染俯视图，然后将其用作地面的光照图？这样，将得到运动的阴影，看起来非常真实。可能性是无限的。

14.8 总 结

本章的内容非常棒，只需使用少量的代码便可极大地提高视觉效果。更重要的是，本章表明，摆脱条条框框的束缚，以非常规方式思考的威力有多么强大。

我们知道，使用面缓存技术可以实现令人惊讶的特殊效果：阴影、光照映射、动画等。请牢记这种技

术，看其是否适用于游戏和引擎的其他方面，也就是说，以逆向思维、旁门左道或非常规方式行事。

最后，本章将我们的注意力转移到了游戏开发中另一个比较微妙的方面——资源（asset）的创建和工具的使用。作为游戏程序员，必要时必须编写代码，但也别忘了可以让艺术师和工具离线完成某些工作。如果艺术师能够事半功倍地实现同样的效果，谁会在乎有些东西从数学上说是否正确呢？重要的是，知道什么时候该玩技巧，什么时候该真刀真枪。

第五部分

高级动画、物理建模和优化

第 15 章：3D 角色动画、运动和碰撞检测

第 16 章：优化技术

第 15 章　3D 角色动画、运动和碰撞检测

本章讨论如何加载和显示 Quake II .MD2 文件格式的 3D 动画模型。介绍一些 3D 运动技术，概述基本的碰撞检测技术，用于检测模型之间以及模型和环境之间的碰撞，具体内容如下：

- 新的库模块；
- 3D 动画简介；
- 加载 Quake II .MD2 文件；
- 使用 Quake II .MD2 文件制作动画；
- 旋转运动和线性运动；
- 参数化曲线运动；
- 使用脚本来实现运动；
- 3D 碰撞和地形跟踪技术。

15.1　新的游戏引擎模块

同样本章的函数很多，可将其作为一个独立的库模块。该库模块名为 T3DLIB13.CPP|H，因此要编译本章的程序，需要程序的主.CPP 文件、DirectX 文件以及下述新的库模块：

- T3DLIB13.CPP——支持 3D 角色的动画 C/C++源代码；
- T3DLIB13.H——头文件。

当然，还需要链接库模块 T3DLIB1-12.CPP|H。

15.2　3D 动画简介

本章的重点是动画，着重介绍角色动画。作者本打算还介绍骨架（skeletal）动画和运动文件（如 Biovision 等），但由于篇幅有限，只能割爱。当前，我们只能将就地使用关键帧动画和插值。关键帧动画的效果很不错，如果它能满足 Quake 和 Quake II 的要求，就能满足我们的实验性用途（experimental use）。本章重点介绍 3 个与动画相关的主要主题：定义、加载和渲染 Quake II .MD2 模型；3D 模型和物体的运动和定位；用于表示角色的 3D 模型的碰撞检测技术。

本章的大部分代码都与第一个主题相关，即加载和渲染基于 id Software 的 Quake II .MD2 格式的 3D 模型。这些内容是本章中最复杂的。对于本章的其他主题，只从理论上进行探讨，重点是理念而不是实现，因为这些没有什么可说的了——前面从不同的角度介绍了如此多的技术，从现在开始，使用什么样的技术来实现完全由读者决定。

15.3　Quake II .MD2 文件格式

Quake II .MD2 文件格式（以下称.MD2）是 id Software 公司为其轰动一时的游戏 Quake II（如图 15.1 所示）开发的。Quake II 将硬件加速带入了游戏开发领域的最前沿，向世界展示了技术的强大威力。Quake II 终结了商业游戏产品使用软件加速的时代，它利用了众多精彩的技术，其中之一是通过关键帧动画和插值实现了更精彩的角色动画。

虽然.MD2 文件没有骨架，也不支持逆向/前向运动学（inverse/forward kinematics），但它分析和理解起来相对比较容易。.MD2 文件优越的地方在于，在 Internet 上，这样的文件数不胜数，还有无数处理这种文件的工具。因此，读者不用愁找不到模型、工具和其他应用程序。

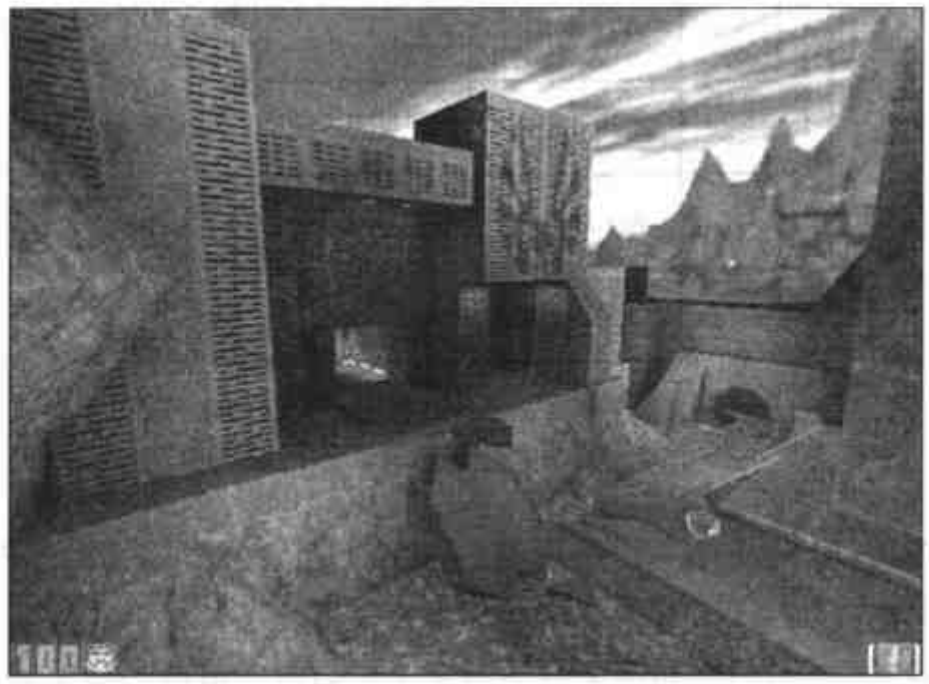


图 15.1　Quake II 运行时的屏幕截图

注意：一个寻找 Quake II 模型的好去处是 Planet Quake，其网址为：

http://www.planetquake.com/polycount/

还有很多可用于处理.MD2 文件的工具，很多这样的工具可在下述网址中找到：

http://www.planetquake.com/polycount/resources/quake2/tools.shtml

最后，这种文件格式的技术规范可在下述网址中找到：

http://www.planetquake.com/polycount/resources/quake2/q2frameslist.shtml

作者从 Planet Quake 下载了一些模型，它们都是免费的——只要您不用来渔利。列出了每个模型的开发者，如果读者想将模型用于商业游戏中，可与开发者联系，并提出请求。接下来介绍这种文件格式本身。

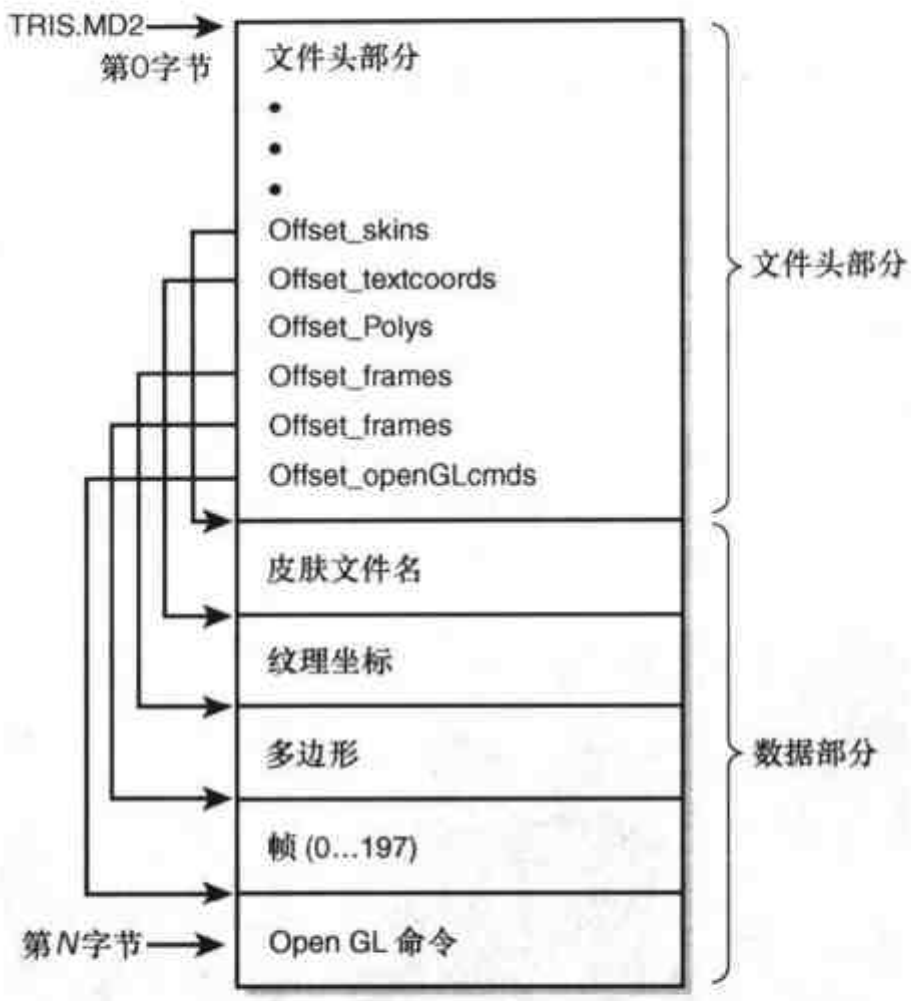
在 Internet 查找有关.MD2 格式的正规描述时，作者遇到了从未遇到的麻烦。我阅读讨论.MD2 格式的每篇文章和每本书籍，在每个范例中，文章的作者总会省略一些重要的细节，因此这里将尽可能详细地介绍这种文件格式，以免读者遇到迟迟不能解决的问题。

我们首先做简要的描述，然后详细探讨其中的细节。.MD2 文件由两个主要部分组成：文件头（header）和数据，如图 15.2 所示。

- 文件头部分——包含模型描述、多边形数、顶点数和动画细节等。
- 数据部分——组成模型的多边形网格的数据，包括所有的多边形、顶点和纹理数据。

提示：在.MD2 文件中，所有的多边形都是三角形，且没有其他的实体。

文件头是标准的文件头，没有什么特别的东西；但数据很有趣：顶点数据定义的不是一帧而是很多帧动画。事实上，每个.MD2 文件包含 198 帧动画，编号为 0～197。



注意：每个子部分的顺序不一定如此

图 15.2　Quake II .MD2 文件的基本结构

警告：有些模型没有遵循最初的规范，包含 199、200 甚至更多帧动画。然而 id Software 最初的规范要求包含 198 帧。仅对动画帧来说这才有关系，因为索引被用来定义组成步行、奔跑、死亡等动作的帧。

这些帧被解释为由一组模型空间中的顶点组成的多边形网格；模型空间为右手坐标系，原点为(0，0，0)，顶点的环绕方向为逆时针（这在后面将变得非常重要，请牢记）。这些顶点决定了每个动画帧中网格的形状。每个具体的动作（如步行、奔跑、跳跃等）都可能由多帧组成。有关这些将在后面详细介绍，但就现在而言，只需牢记文件开头有一个定义了模型的文件头，根据文件头信息，可以加载帧数据，它们是纯粹的二进制流（binary stream）。

动作是通过顺序播放一系列动画帧实现的。例如，动作“站着不动（standing idle）”由第 0～39 帧组成，因此通过顺序播放这些帧，可以实现“站着不动”的动作。然而，这些帧是关键帧，这意味着如果愿意，可以通过线性插值在这种帧之间计算出其他帧，使动画更为流畅。例如，您可能决定在第 i 和 i+1 帧之间插入 3 帧，为此，将分别使用权重值 0.25、0.50 和 0.75，根据这两帧中顶点的位置计算出插入帧的顶点位置。

接下来的一项重要内容是纹理图，id Software 的技术人员喜欢将其称为皮肤（skin）。图 15.3 是一个.MD2 模型中的标准皮肤纹理，它是 256 色的，大小为 256×256，文件格式通常为.PCX。我们必须将所有的皮肤转换为 16 位颜色和.BMP 格式的。

皮肤的意思是，它被用作模型的纹理。每个模型都可能有很多皮肤，颜色和样式各不相同。有趣的是，皮肤并非.MD2 文件的组成部分，在.MD2 文件中，通过文件名来指定皮肤，这种文件位于 Quake II 游戏目录中。在分析.MD2 文件的文件头时，可能发现模型有 3 种皮肤（也许分别为蓝色、梅彩色和血红色）。在这种情况下，您将根据路径和名称来加载.PCX 文件（即纹理）。



图 15.3　一个 Quake II 皮肤

15.3.1　.MD2 文件头

下面详细介绍.MD2 文件格式的文件头部分。首先，Internet 上的大部分.MD2 模型经过解压缩后，都将位于同一个目录中，其中包含大量与模型相关的文件，如武器、美工（art）和声音。我们感兴趣的是模型和皮肤，这里重点介绍它们。从 Internet 下载模型或查看附带光盘中的模型时，读者将发现目录中至少包含以下内容：

- TRIS.MD2——包含角色的模型数据的文件。
- WEAPON.MD2——可能有也可能没有，但通常有。该文件是武器模型，也是.MD2 文件。本章不打算介绍武器模型，不过其加载方法与角色模型类似。

接下来将看到很多纹理文件，它们为 256 色，大小为 256×256，文件格式为.PCX，可以使用任何绘图程序（Paint Shop Pro、Photoshop 等）查看它们。这些图像是皮肤，被用作映射到模型上的纹理。之所以要求它们是 256×256 的，是因为在.MD2 文件的数据部分，纹理坐标 u 和 v 的取值范围为 0～255。然而，要使用这些纹理，必须将其转换为 16 位颜色模型，并保存为.BMP 文件（虽然我们也有.PCX 加载函数，但.BMP 更容易）。

另外，还有一些用作图标的小型图像以及用作音效的.WAV 文件。对于这些文件，我们也不感兴趣。我们真正需要的是 TRIS.MD2 文件以及一种或多种皮肤（将其转换为 16 位.BMP 格式）。在大多数情况下，作者从 Polycount 或其他网站下载.MD2 模型，然后将其解压缩到一个目录中。然后查看目录中的图像，从中找出两三种喜欢的皮肤，将其色深增加到 16 位，亮度提高 10～20，再保存为.BMP 文件。文件名无关紧要，因为我们将修改文件头中内置的纹理皮肤的文件名和路径。这是可行的，因为所有的纹理坐标都保持不变（u、v 的取值范围仍为 0～255），纹理也没变，因此不会出现异常。

提示：Quake II .EXE 文件在其所在的目录中查找纹理，因此我们不从路径中提取文件名，而是给.MD2 加在函数提供纹理的.BMP 版本的文件名。

当然，我们不能提供其他皮肤，而只能使用模型指定的皮肤，这样纹理坐标才能与皮肤匹配，进而被正确地映射到模型上。然而，至少可以通过将纹理放在希望的地方，并使用喜欢的名称，来简化纹理访问。

介绍过下载.MD2 文件可在硬盘中找到的文件后，来看看文件头部分和数据部分。首先，请注意.MD2 文件是二进制的，必须以字节为单位进行读取。下面是文件头部分，它总是位于.MD2 文件的开头，从第 0 字节开始：

```
// 这是 Quake II .MD2 文件使用的文件头结构
// 作者对其中的字段进行了注释，以方便读者理解
typedef struct MD2_HEADER_TYP
{
int identifier;   // 指出文件类型，应为 IDP2
int version;     // 本版号，应为 8
int skin_width;    // 用作皮肤的纹理图的宽度
int skin_height;  // 用作皮肤的纹理图的高度
int framesize;   // 每个动画帧的大小，单位为字节
int num_skins;   // 总皮肤数
int num_verts;   // 每帧中的顶点数，所有帧的顶点数都相同
int num_textcoords; // 整个文件中的纹理坐标数，可能多于每帧的顶点数
int num_polys;   // 每个动画帧中的多边形数
int num_openGLcmds; // openGL 命令数，这种命令用于优化渲染，我们不使用
int num_frames;   // 动画帧数

int offset_skins;   // 皮肤数组离文件开头的距离，单位为字节
            // 皮肤数组中存储了皮肤的文件名
            // 每个文件名的长度为 64 字节
int offset_textcoords; // 纹理坐标数组离文件开头的距离，单位为字节
int offset_polys;   // 多边形列表离文件开头的距离，单位为字节

int offset_frames;   // 顶点数组离文件开头的距离，单位为字节
int offset_openGLcmds; // OpenGL 命令数组离文件开头的距离，单位为字节

int offset_end;    // 文件开头到文件末尾的距离，单位为字节

} MD2_HEADER, *MD2_HEADER_PTR;
```

在不同的文献中，上述字段名可能稍有不同，但大多数人都尽可能使用 id Software 最初定义的名称。不过，字段的数据类型和排列顺序必须与上述结构中相同，否则当您加载 TRIS.MD2 文件时，将无法参考文件头中的数据。下面详细介绍每个字段：

- int identifier——这是一个幻数（magic number）字段，指出文件类型为.MD2，其值必须是 little endian 格式的“IDP2”。换句话说，在使用 Intel 处理器（采用 big endian 格式）的 PC 上，应查找下述值（I 位于最右边）：

```
#define MD2_MAGIC_NUM (('I') + ('D' << 8) + ('P' << 16) + ('2' << 24))
```

- int version——版本号，总是为 8。同样，作者定义了一个用于比较版本号的常量：

```
#define MD2_VERSION   8
```

如果幻数或版本不正确，则要么.MD2 文件被损坏，要么计算机出了问题。

- int skin_width——用作模型皮肤的纹理图的宽度，单位为像素。对于第三方创建的模型，这通常为 256；但也可能为其他值。如果要确保加载函数的健壮性，应查看这个值。
- int skin_height——用作模型皮肤的纹理图的高度，单位为像素。对于第三方创建的模型，这通常为 256；但也可能为其他值。如果要确保加载函数的健壮性，应查看这个值。

在同一个模型中，所有用作皮肤的纹理图大小都相同，因此查看上述两个字段的值后，便可知道所有纹理的大小。在 99%的模型中，上述两个字段的值都是 256。

- int framesize——模型中单个动画帧的大小，单位为字节，如图 15.4 所示。每个帧都由帧头和模型中每个顶点的信息组成。稍后将详细介绍，就现在而言，读者只需牢记每个帧都由帧头和模型的所有顶点组成，帧头指出了如何对顶点进行定位和缩放。遵循 id Software 规范的.MD2 文件通常包含 198 帧。

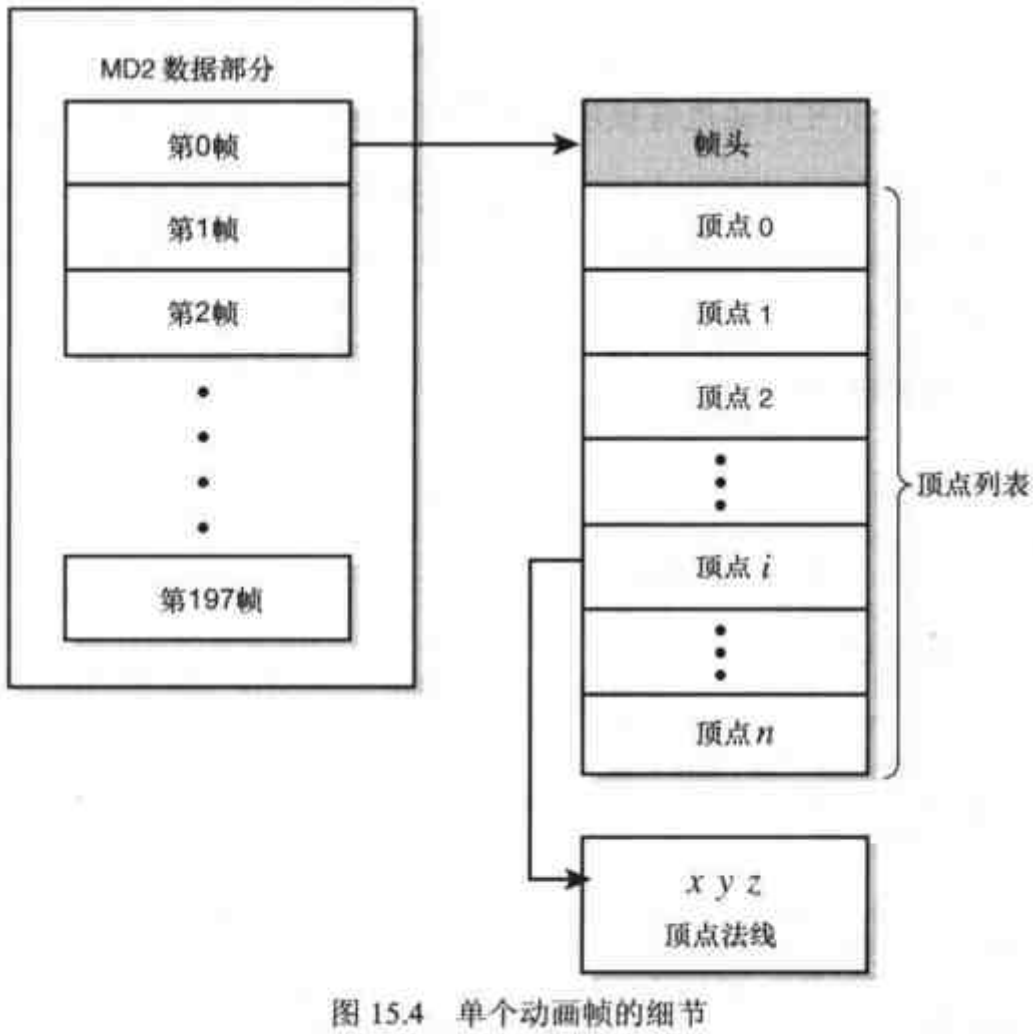


图 15.4　单个动画帧的细节

- int num_skins——这个字段指出了总皮肤数，皮肤用 ASCII 文件名指定，存储在模型文件所在的磁盘目录中。皮肤名称存储在文件头末尾的一个字符串数组中，每个皮肤名都为 64 字节，不足 64 字节时用空字符（0）填满。模型中可能有一个或多个皮肤名，在大多数情况下，这些名称中都包含 Quake II 专用的路径信息，所以我们将忽略它们，在模型所在的目录中手工查找皮肤并对其重命名，然后在加载模型时，传递一个包含皮肤（16 位颜色模式和.BMP 格式）的路径和文件名的字符串。

● int num_verts——单个模型帧中的顶点数，而不是整个文件中的顶点数。由于模型总是由同样的多边形组成，因此每帧中的顶点数相同，惟一变化的是顶点的位置。这是.MD2 格式实现动画的方式。

● int num_textcoords——模型文件中的纹理坐标总数。读者可能认为，每个顶点都需要一个纹理坐标，但可能并非如此。纹理坐标是纹理空间中的一对 u、v 坐标，很多时候，纹理坐标数可能比顶点数多或少，因为多个顶点可能共用一个纹理坐标。可以将纹理坐标视为模型中的所有动画帧需要的各种不同的纹理坐标。在大多数情况下，只有几百个纹理坐标。另外，只有一个纹理坐标集合，由所有动画帧共享，如图 15.5 所示。

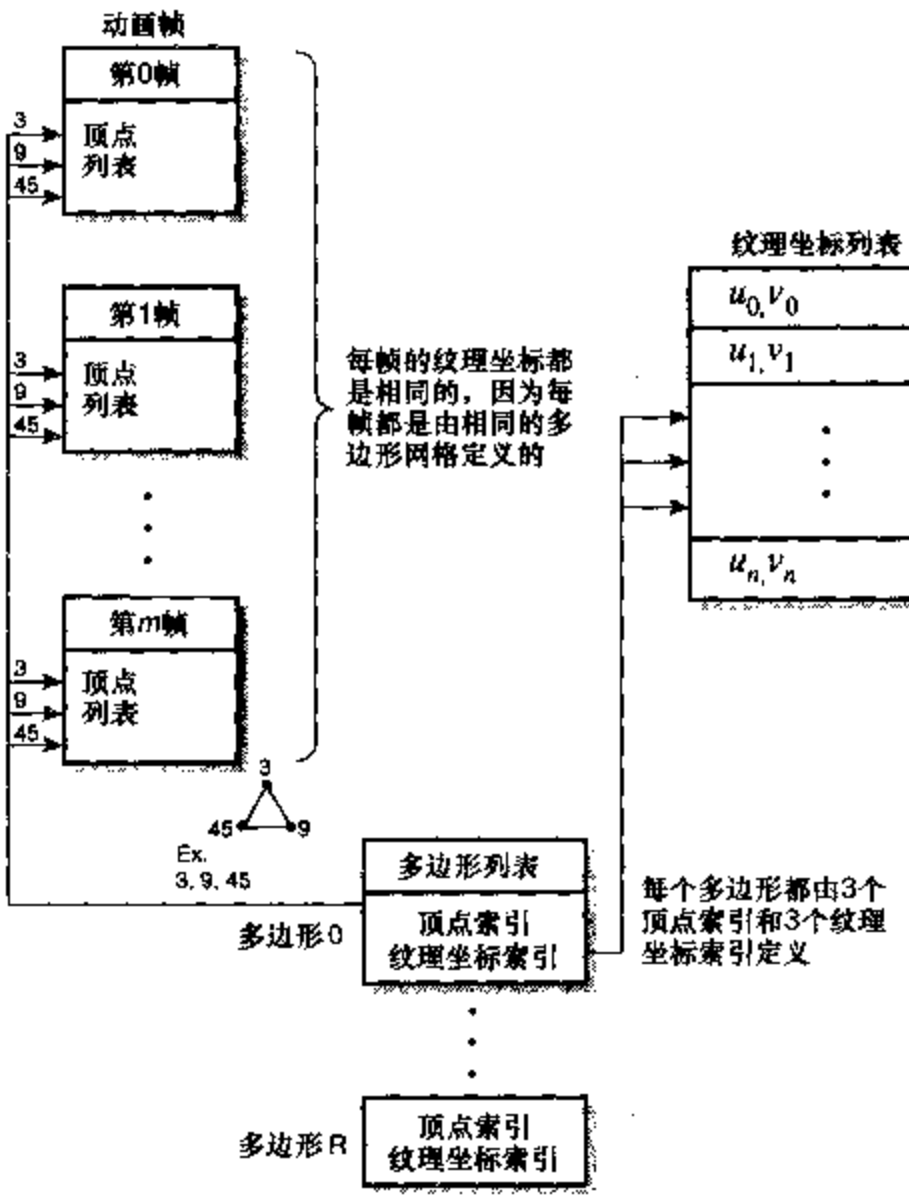


图 15.5　由所有动画帧共享的纹理坐标集合

● int num_polys——模型（每个动画帧）中的多边形数量。每个动画帧中的多边形数量总是相同的，如果模型中有 800 个多边形，则每个动画帧中也有 800 个多边形。

● int num_openGLcmds——这是 OpenGL 命令数，OpenGL 命令有助于优化硬件渲染，我们将忽略这个字段。使用硬件进行渲染时，通过渲染多边形扇或多边形条带可提高性能，OpenGL 命令指示何时以何种方式进行渲染，渲染多少个等。

● int num_frames——动画帧总数，大多数.MD2 模型为 198。

接下来的几个字段内存偏移量，供读取数据时使用。由于.MD2 文件是二进制的，因此提供这些字段是有道理的。这些字段指出要读取某项数据时，应从.MD2 文件的开头开始跳过多少个字节。然后声明一个包含一个元素的顶点数组，并使用索引来访问其他数据，如图 15.6 所示。

● int offset_skins——指出包含皮肤文件名的皮肤数组离文件开头有多远，单位为字节。每个文件名都是 64 字节。

● int offset_textcoords——指出纹理坐标数组离文件开头有多远，单位为字节。每个纹理坐标都是 SHORT，其定义如下：

```
// 一对 u、v 纹理坐标
```

```
typedef struct MD2_TEXTCOORD_TYP
{
short u,v; // 纹理坐标
} MD2_TEXTCOORD, *MD2_TEXTCOORD_PTR;
```

因此，每个纹理坐标都是 32 位（4 字节），我们根据 num_textcoords 和上述结构来存取纹理坐标。

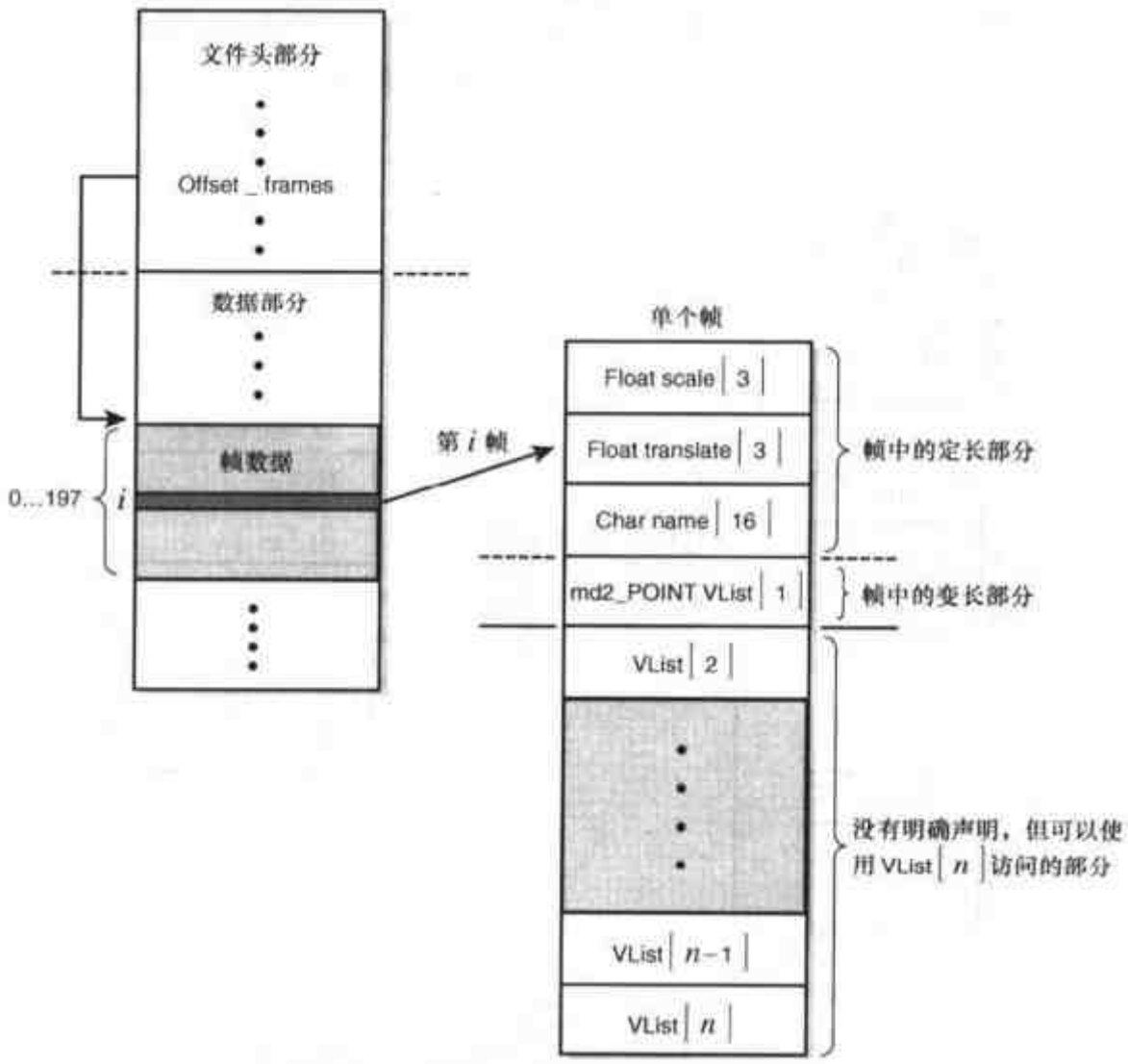


图 15.6　声明一个变长的数据结构模板

● int offset_polys——指出多边形网格数据离文件开头有多远，单位为字节。每个多边形都是用一组顶点索引定义的，这些索引指向组成该多边形的顶点。如果多边形的顶点索引为(12，56，99)，则无论在哪一帧（顶点集）中，该多边形的顶点索引都是(12，56，99)，因此在模型中，对于多边形与顶点索引之间的关系，只需定义一次。显示动画时，我们不断地更换顶点集，但在每个顶点集中，顶点索引保持不变，如图 15.7 所示。

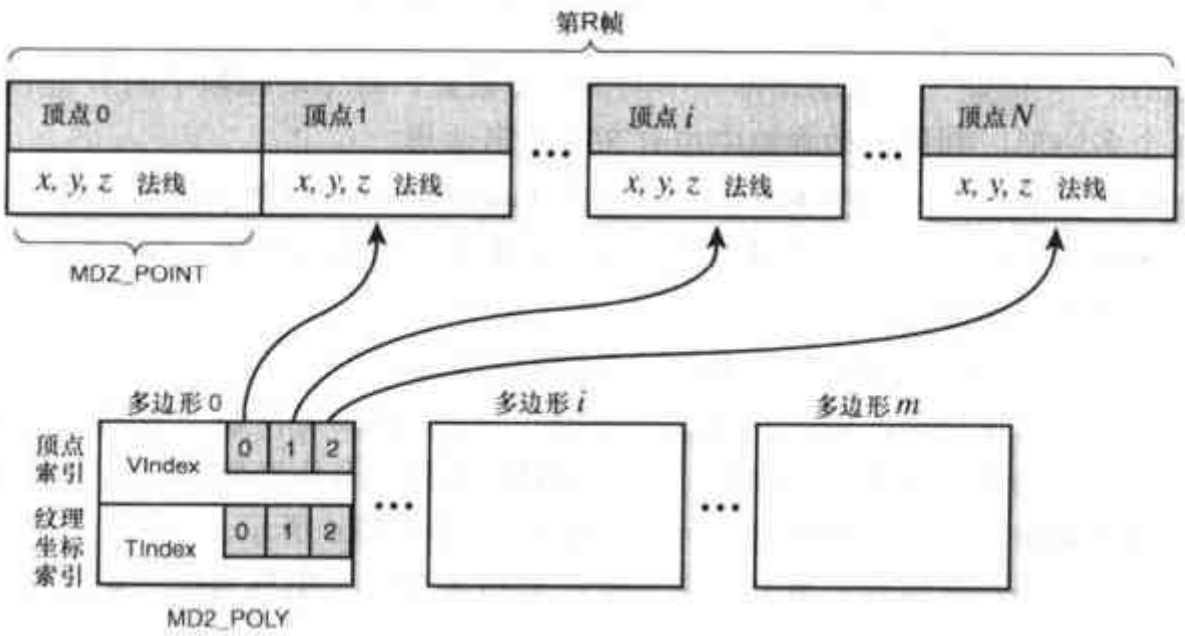


图 15.7　多边形和顶点之间的关系

表示多边形的数据结构如下：

```
// 表示 md2 多边形（三角形）的数据结构
```

```
// 由 3 个顶点索引和 3 个纹理坐标索引组成
typedef struct MD2_POLY_TYP
{
unsigned short vindex[3];  // 顶点索引
unsigned short tindex[3];  // 纹理坐标索引
} MD2_POLY, *MD2_POLY_PTR;
```

多边形列表是一个多边形结构数组。每个多边形结构有 6 个元素，前 3 个为 unsigned short（无符号短整数），指定了顶点索引；后 3 个也是 unsigned short，指定了纹理坐标索引。因此，多边形是间接（indirectly）定义的，如图 15.8 所示。

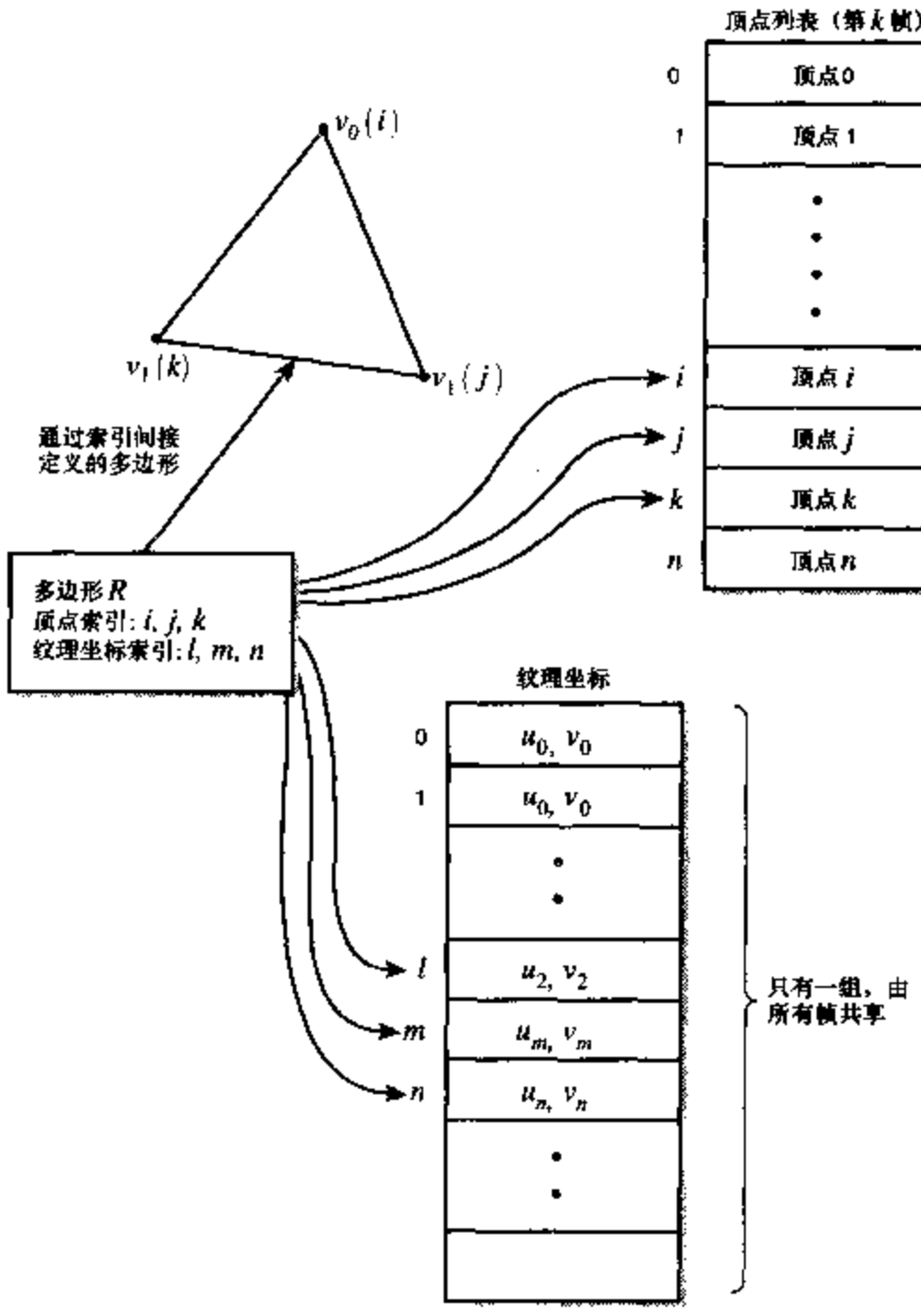


图 15.8　每个多边形都是间接定义的

● int offset_frames——这是帧数据离文件开头的距离，单位为字节，如图 15.9 所示。每个帧由小型帧头和定义该帧的顶点组成。

用于表示帧的数据结构如下：

```
// 表示 MD2 帧的数据结构
// 帧头中包含如何对顶点进行缩放和平移的信息
// 接下来是包含顶点数据的数组，数组的长度是可变的，因此将数组长度定义为 1
// 这样编译器将允许存取第 n 个元素
typedef struct MD2_FRAME_TYP
{
float scale[3];     // 顶点的 x、y、z 的缩放因子
float translate[3];  // 顶点的 x、y、z 的平移因子
char name[16];      // 动画帧的 ASCII 名称
MD2_POINT vlist[1];  // 顶点数组的第一个元素
```

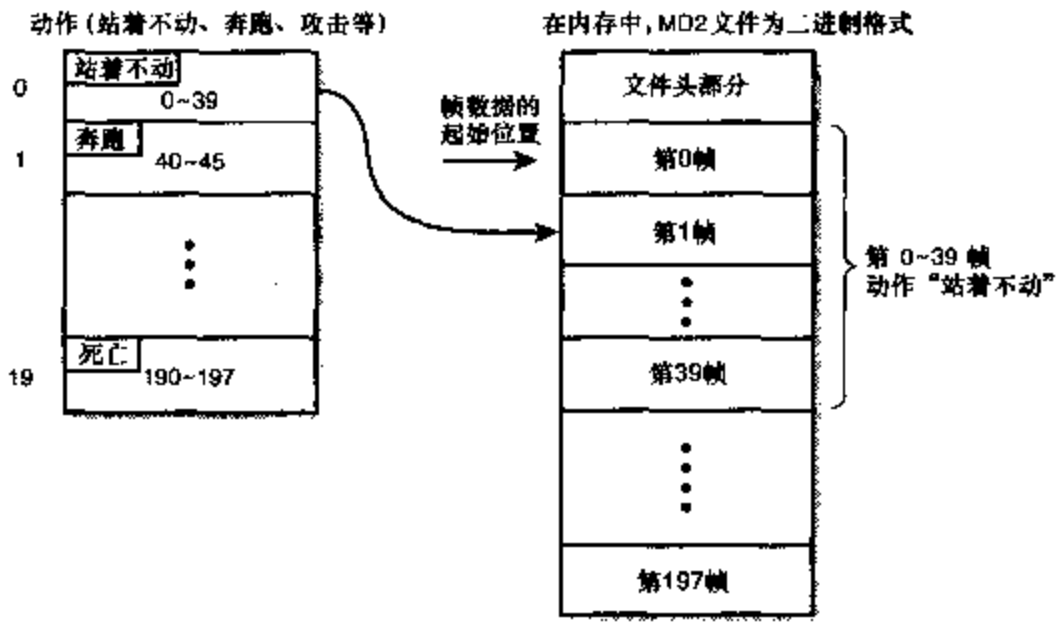


图 15.9　动画帧数组

帧头的后面是顶点数据数组，稍后再介绍，现在先介绍一下帧头。帧头有 4 个字段，第一个是 float scale[3]。

- float scale[3]——一个包含 3 个元素的 float 数组，它们指定了 x、y、z 缩放因子，用于对帧中的每个顶点进行缩放，因此加载.MD2 帧时，必须据此对其中的每个顶点进行缩放。x 缩放因子存储在 scale[0]中，y 缩放因子存储在 scale[1]中，z 缩放因子在 scale[2]中。缩放因子的用途有两个：首先，为节省存储空间，每个顶点坐标被压缩为 8 位，因此必须将其放大到正常值；其次，帮助实现动画——要对动作中的帧进行缩放，可以使用缩放因子。

- float translate[3]——这是一个包含 3 个元素的 float 数组，它们指定了 x、y、z 平移因子，用于对帧中的每个顶点进行平移，因此加载.MD2 帧时，必须据此对每个顶点进行平移。x 平移因子存储在 translate[0]中，y 缩放因子存储在 translate[1]中，z 缩放因子存储在 translate[2]中。同样，平移因子的用途也是节省存储空间，因为每个顶点坐标被压缩为 8 位，但更重要的是，用于在模型空间中移动每个帧，以实现动画。例如，“跳跃”动作就是这样实现的——平移模型坐标（而不是平移世界位置），并执行局部坐标到世界坐标变换。

图 15.10 说明了帧头指定的要对每个顶点进行的变换。给定顶点 v，下面是执行这种变换的伪代码：

```
v.x = v.x * scale[0] + translate[0];
v.y = v.y * scale[1] + translate[1];
v.z = v.z * scale[2] + translate[2];
```

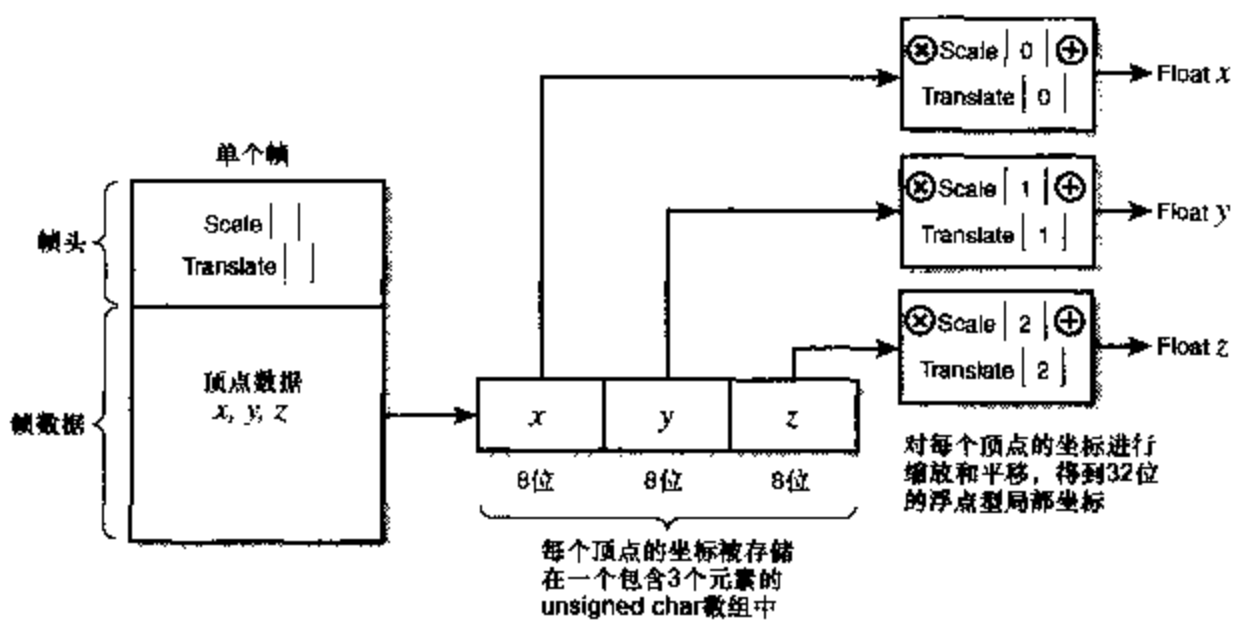


图 15.10　将帧头中的变换信息用于每个顶点

- char name[16]——这个字段是帧的 ASCII 名称，如“Terminator T1000”。

最后，下面是实际的帧数据：

● MD2_POINT vlist[1]——这是一个指向顶点数组的指针。然而，我们并不知道有多少个顶点，因此将该数组的大小定义为 1。如果您访问元素 vlist[2]等，编译器将允许这样做，但必须有这样的空间，否则将发生存取违规。这是一种常用的定义变长数据结构的技巧：声明一个指针或数组，然后在知道其长度的情况下使用索引来访问。然而，可以使用单个数据结构模板来表示任何长度的数组。因此，顶点列表 vlist[ ]包含一个动画帧中所有的顶点。每帧中总是包含 num_verts 个顶点，用于表示顶点的数据结构如下：

```
// 用于表示顶点的数据结构，包含 x、y、z 坐标（各占 8 位）和一个顶点法线索引
// 该索引用于从 id software 提供的法线查找表中查找索引
typedef struct MD2_POINT_TYP
{
unsigned char v[3];      // x、y、z 坐标，都被压缩为 1 个字节
unsigned char normal_index; // 指向 id Software 提供的索引查找表的索引（没有使用）
} MD2_POINT, *MD2_POINT_PTR;
```

注意，每个顶点的 x、y、z 坐标都是 unsigned char，即只有 8 位。现在，读者明白了为何要在帧头中提供缩放和平移支持吧。顶点由 x、y、z 坐标（共 3 字节）和 normal_index（1 字节）组成，后者用于支持另一种压缩机制。在每个帧中，每个顶点都有顶点法线，如图 15.11 所示。

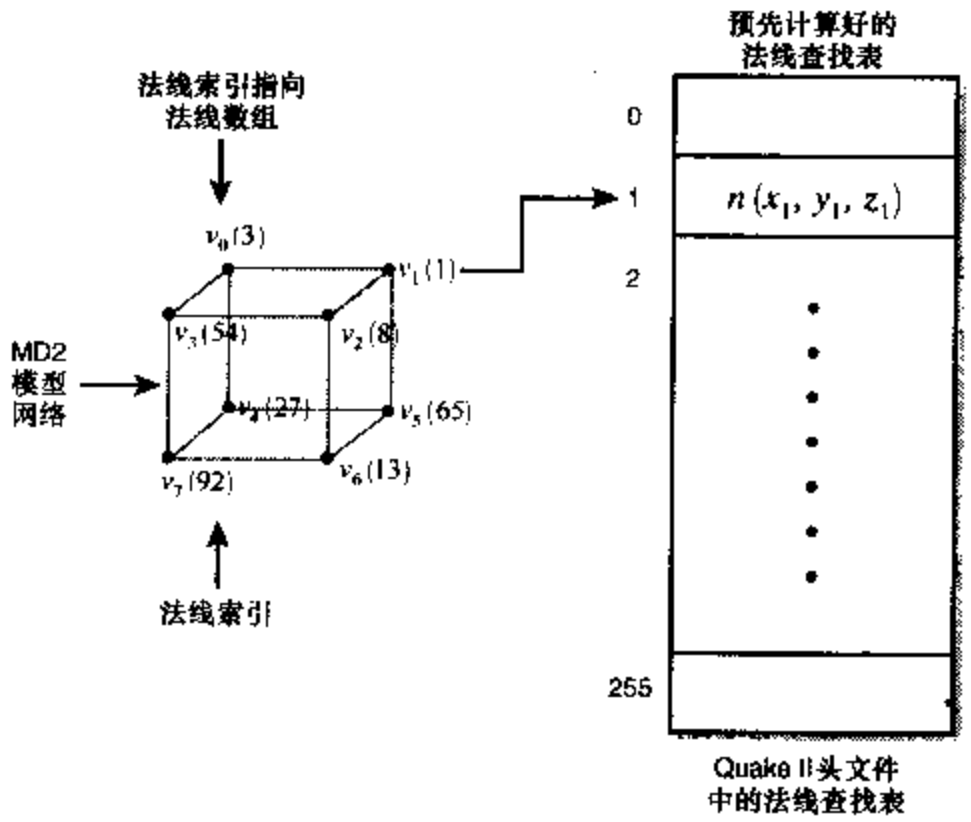


图 15.11　Quake II 中用于光照计算的法线

同样，法线也是基于模型空间坐标系来表示的，但在.MD2 文件中，使用预先计算好的法线数组中索引来指定法线。id Software 创建了一个法线查找表，这个表位于 Quake II 源文件中，其中存储了每个索引对应的法线。虽然我们不使用顶点法线（我们将计算多边形的面法线），但这种优化和间接定义可节省存储空间和提高计算速度。

总之，帧数据是一个包含 198 个元素的帧数组，其起始位置由 offset_frames 指定。每个帧由帧头和顶点数组组成。帧头由缩放因子、平移因子和帧的 ASCII 名称组成，后面是帧的顶点数据。每个顶点由 4 个 1 字节的值组成，其中前 3 个是顶点在模型空间中的 x、y、z 坐标，最后一个指向顶点法线查找表的索引（我们不支持），该查找表包含 256 个元素。

● int offset_openGLcmds——OpenGL 命令离文件开头的距离，单位为字节。这里不使用 OpenGL 命令，因此不考虑这个字段。

● int offset_end——文件末尾到文件开头的距离，单位为字节。这个字段是不必要的，因为我们可以向文件系统查询文件的长度。

下面来编写加载.MD2 文件的代码。

15.3.2 加载 Quake II .MD2 文件

读者可能还记得，我们最初打算使用 OBJECT4DV2 来支持动画，因此在该数据结构中加入了支持多帧的功能。然而，作者后来想，OBEJCT4DV2 只是一个用于渲染网格的容器类，为何让它支持动画、.MD2 文件呢？一种更好的方法是，编写一个提取函数，使用它提取多边形数据，并用这些数据来“填充”一个 OBJECT4DV2，然后将其传递给渲染引擎，这样便可以利用以前编写的渲染函数了。这就是作者最终决定使用的方法。

加载并渲染.MD2 的总体进攻计划如下：

1．加载.MD2 文件，将其转换为一种更易于使用的格式，以便能够从中提取帧，并将其加载到 OBJECT4DV2 中。

2．编写两个支持函数，以便 OBJECT4DV2 能够用于存储.MD2 帧。

3．编写一个提取函数，从.MD2 模型动画中提取指定的帧，将其加载到一个 OBJECT4DV2 中，然后利用已有的引擎像以前那样对 OBJECT4DV2 进行处理。

最初，作者打算将.MD2 文件加载到内存中，然后从中直接提取帧数据，将其加载到 OBJECT4DV2 中，但这意味着需要执行顶点数据的缩放、平移等运算。另外各种压缩和间接数据结构也使得这是一种糟糕的主意。一种更好的方法是，创建另一个适合存储.MD2 数据的容器类（数据结构），并在其中使用已有的表示点、顶点、纹理等的数据结构。这样，访问这个容器类并从中提取帧数据时，速度将非常快。

这里的重点是速度。我们希望任何帧的渲染都能够几乎瞬时完成，同时在不同帧之间切换的开销不多。另外，设计容器类时，还需要有些远见，使其能够支持动画、运动和通用状态信息，就像一个 blitter 对象（BOB）。最后，作者设计的数据结构（容器类）如下：

```
typedef struct MD2_CONTAINER_TYP
{
int state;      // 模型的状态
int attr;       // 模型的属性
int color;      // 没有纹理时的颜色
int num_frames;   // 帧数
int num_polys;   // 多边形数
int num_verts;   // 顶点数
int num_textcoords; // 纹理坐标数

BITMAP_IMAGE_PTR skin;  // 指向纹理的指针

MD2_POLY_PTR polys; // 指向多边形列表的指针
VECTOR3D_PTR vlist; // 指向顶点坐标数组的指针
VECTOR2D_PTR tlist; // 指向纹理坐标数组的指针

VECTOR4D world_pos; // 物体的位置
VECTOR4D vel;     // 物体的速度

int  ivars[8];   // 整型变量
float fvars[8];   // 浮点型变量
int counters[8];  // 通用计数器
void *link;       // 通用 void 指针
int anim_state;      // 动作状态
int anim_counter;  // 通用动作计数器
int anim_mode;   // 播放一次还是循环播放
int anim_speed;   // 值越小，动作切换速度越快
int anim_complete; // 指出动作是否播放完毕的标记
float curr_frame;   // 当前帧

} MD2_CONTAINER, *MD2_CONTAINER_PTR;
```

对所有字段都做了清晰的注释，其中很多字段与.MD2 文件头中相同。为确保读者和作者对这些字段的认识一致，下面简要地介绍一下：

- int state——这是模型的状态信息，如活着、死亡等。
- int attr——这是模型中多边形的属性，即想如何渲染多边形（Shader 属性）。之所以需要这个字段，

是因为.MD2 文件中没有任何有关着色和光照处理的信息，因此我们需要指定如何渲染模型。这个字段用于存储这些属性，将帧数据插入到 OBJECT4DV2 中时，将相应地设置每个多边形的这些属性。在大多数情况下，该字段的值如下：

```
(POLY4DV2_ATTR_RGB16 | POLY4DV2_ATTR_SHADE_MODE_FLAT |
POLY4DV2_ATTR_SHADE_MODE_TEXTURE)
```

即使用 16 位颜色模式、恒定着色，支持纹理映射。

- int color——没有纹理时多边形的颜色。在大多数情况下，将其设置为 RGB(255，255，255)，让纹理的反射率最高。
- int num_frames——.MD 模型中的总帧数，通常为 198。这个值直接从.MD2 文件头中复制而来。
- int num_polys——单个动画帧中的多边形数量，取值与.MD2 文件头中相应的字段相同。由于多边形网格为模板，因此每个动画帧中的多边形数量都相同，在每个帧中，惟一变化的是顶点的位置。
- int num_verts——单个动画帧中的顶点数（通常有数百个），每个动画帧的顶点数都相同，惟一变化的是顶点的位置。
- int num_textcoords——可供所有顶点使用的纹理坐标数。在每个动画帧中，使用纹理坐标数组的索引来指定纹理坐标。在大多数情况下，一个.MD2 模型中有两三百个纹理坐标。
- BITMP_IMAGE_PTR skin——指向模型使用的纹理的指针。这个位图是这样得到的：加载模型目录中 256 色、256×256 的皮肤，将其色深增加到 16 位，再保存为.BMP 文件。该位图将被用作纹理，映射到模型上。

下面的字段指定了几何数据的存储位置，但不同于.MD2 文件中，这些几何数据可直接用于渲染——经过了缩放和平移等处理。另外，现在所有的几何数据都是用本书前面定义的数据结构存储的，因此处理起来很容易。

- MD2_POLY_PTR polys——这是一个指向多边形列表的指针。多边形列表是一个结构数组，每个结构都包含一个多边形的顶点索引和每个顶点的纹理坐标索引，就像.MD2 文件中一样。它基本上是一个间接数组，没必要与.MD2 文件中不同。
- VECTOR3D_PTR vlist——这是一个指向顶点坐标列表的指针。每个顶点都是一个 VECTOR3D。这个数组中包含所有动画帧中每个顶点的坐标，数据的存储顺序如图 15.12 所示，即依次为第 0 帧、第 1 帧...第 n 帧。每帧都有 num_verts 个顶点，每个顶点都是 VECTOR3D。要访问某个帧中的顶点，必须将该帧的编号和 num_verts 的乘积用作基索引（base index）。
- VECTOR2D_PTR tlist——这是一个指向纹理坐标列表的指针。每个纹理坐标都以浮点格式存储在一个 VECTOR2D 中。纹理坐标列表是一个数组，包含 num_textcoords 个元素。

下面的两个字段用于在 3D 空间中对模型进行移动和定位：

- VECTOR4D world_pos——用世界坐标表示的模型位置。

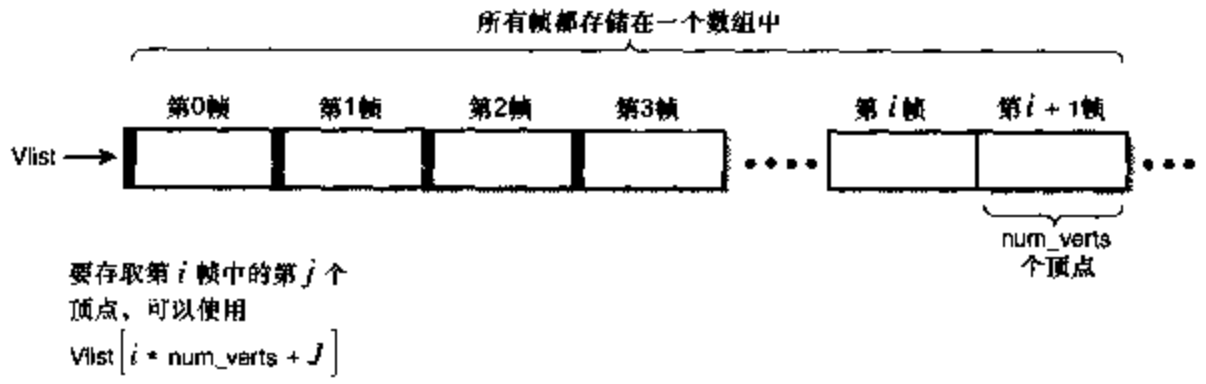


图 15.12　顶点列表

- VECTOR4D vel——模型的速度。

下面的字段用于存储状态信息、执行逻辑等：

- int ivars[8]——整型变量。
- float fvars[8]——浮点型变量。
- int counters[8]——计数器。

- void *link——void 指针。

下面的字段很重要，用于帮助播放动画和跟踪模型状态：

- int anim_state——动画状态，这里指的是当前播放的是哪个动作。id Software 的开发人员拿出了一个要在每个模型中支持的动作清单，包括步行、奔跑、死亡、射击等。然而，在过去的几年中，很多人得到了创造性许可，在这个清单中加入了更多的动作，因此作者不得不查看 Quake II 源代码本身，以了解有多少动作以及这些动作的名称和帧索引，结果如下：

—.MD2 文件总共有 198 帧；

—有 20 种不同的动作；

—每个动作由多个帧组成，它们的帧索引总是相同的。

表 15.1 列出了这些动作及其名称和帧索引。这是作者通过查看 Quake II 源代码和头文件 M_PLAYER.H 后获悉的，这些文件可以在 http://www.fileplanet.com/files/80000/83000.shtml 中找到。

如果上述链接无效，可通过 Google.com 或 Yahoo.com 进行搜索。

表 15.1　　MD2 文件格式动画帧的动作规范

动作编号	文本名称	帧索引	动作编号	文本名称	帧索引
0	STANDING_IDLE	0–39	10	WAVE	112–122
1	RUN	40–45	11	POINT	123–134
2	ATTACK	46–53	12	CROUCH STAND	135–153
3	PAIN 1	54–57	13	CROUCH WALK	154–159
4	PAIN 2	58–61	14	CROUCH ATTACK	160–168
5	PAIN 3	62–65	15	CROUCH PAIN	169–172
6	JUMP	66–71	16	CROUCH DEATH	173–177
7	FLIP	72–83	17	DEATH BACK	178–183
8	SALUTE	84–94	18	DEATH FORWARD	184–189
9	TAUNT	95–111	19	DEATH SLOW	190–197

大部分文本名称都描述了动作的含义。例如，Pain 1、Pain 2 和 Pain 3 是模型被打中后感到痛苦的动作；ATTCK 是模型开火的动作。然而，最重要的数据是帧数，它们不会变化。问题是，Internet 上的很多模型都是不正确的，有些动作的帧数更多或更少。这不是什么大问题，但您迟早需要解决，否则动作将有问题。最好的解决方法是，使用一个可设置的帧索引表，这样必要时可以根据模型动态地修改它。作者定义了一组可用于设置 anim_state 的常量：

```
// md2 动画状态
#define MD2_ANIM_STATE_STANDING_IDLE  0 // 站着不动
#define MD2_ANIM_STATE_RUN            1 // 奔跑
#define MD2_ANIM_STATE_ATTACK         2 // 开火/攻击
#define MD2_ANIM_STATE_PAIN_1         3 // 被打中版本 1
#define MD2_ANIM_STATE_PAIN_2         4 //被打中版本 2
#define MD2_ANIM_STATE_PAIN_3         5 //被打中版本 3
#define MD2_ANIM_STATE_JUMP           6 // 跳跃
#define MD2_ANIM_STATE_FLIP           7 // 做手势
#define MD2_ANIM_STATE_SALUTE         8 // 敬礼
#define MD2_ANIM_STATE_TAUNT          9 // 嘲笑
#define MD2_ANIM_STATE_WAVE           10 // 挥手致意
#define MD2_ANIM_STATE_POINT          11 // 指向别人
#define MD2_ANIM_STATE_CROUCH_STAND   12 // 不动时蹲下
#define MD2_ANIM_STATE_CROUCH_WALK    13 // 行走时蹲下
#define MD2_ANIM_STATE_CROUCH_ATTACK  14 // 蹲下的同时开火
#define MD2_ANIM_STATE_CROUCH_PAIN    15 // 蹲下时被打中
#define MD2_ANIM_STATE_CROUCH_DEATH   16 // 蹲下时死去
#define MD2_ANIM_STATE_DEATH_BACK     17 // 死亡时向后倒
#define MD2_ANIM_STATE_DEATH_FORWARD  18 // 死亡时向前倒
```

```
#define MD2_ANIM_STATE_DEATH_SLOW   19 // 缓慢死去（沿任何方向倒下）
```

注意：读者可能注意到了，其中没有定时信息。基本上，我们让设备来决定播放速度以及是否插入帧；然而，作者加入了这种支持，以简化动画实现工作。

- int anim_counter——用于记录帧数的计数器，该计数器将不断增加，进入下一个动作时清零。
- int anim_mode——指定如何播放 anim_state 指定的动作：播放一次还是循环播放。通常，anim_mode 的取值为下列两个常数之一：

```
// 动作播放模式
#define MD2_ANIM_LOOP       0 // 循环播放
#define MD2_ANIM_SINGLE_SHOT  1 // 播放一次
```

- int anim_speed——动画的实际播放速度，它的值越小，速度越快。
- int anim_complete——指出单次动作是否播放完毕（1 表示完毕，0 表示还没完），用于控制逻辑中。例如，在“跳跃”动作完成后，可能播放着地的声音等。
- float curr_frame——当前渲染的帧。这是一个浮点型变量，而不是整型变量，这很重要。我们将支持在关键帧之间插入其他帧，因此帧编号可以为小数。例如，帧编号 5.4 表示在第 5 帧和第 6 帧之间插入一帧，其中第 6 帧的权重为 40%。本章后面讨论动画和顶点插值时将详细讨论这一点。

下面首先来实现加载函数。我们需要将磁盘中的.MD2 文件加载到内存中，然后读取.MD2 文件头，从中提取所有的信息，将其存储到容器类 MD2_CONTAINER 中，然后释放.MD2 文件占用的内存。对于这样大的函数，作者通常不列出其代码，但这里有充分的理由列出其代码。下面是该函数的代码，其中包含大量的注释，作者删除了大量处理错误和输出的代码，有关该函数的完整代码，请参阅附带光盘：

```
int Load_Object_MD2(MD2_CONTAINER_PTR obj_md2, // 用于存储 MD2 模型数据
      char *modelfile,  //.MD2 模型的文件名
      VECTOR4D_PTR scale, // 初始缩放因子
      VECTOR4D_PTR pos,  // 初始位置
      VECTOR4D_PTR rot,  // 初始旋转角度（没有实现）
      char *texturefile, // 纹理的文件名
      int attr,      // 光照/模型属性
      int color,     // 没有纹理时的颜色
      int vertex_flags)  // 控制顶点排列顺序等
{
// 这个函数加载一个.md2 文件，提取其中所有的数据，将其存储到一个容器类中
// 后面将从该容器类中的帧加载到 OBJECT4V2 中，以便动态地进行渲染

FILE *fp   = NULL; // 指向模型文件的指针
int  flength = -1;  // 用于存储文件长度
UCHAR *buffer = NULL; // 用于存储 md2 文件中的数据

MD2_HEADER_PTR md2_header;  // 指向 md2 文件头的指针


// 首先加载.md2 模型文件
if ((fp = fopen(modelfile, "rb"))==NULL)
  {
  Write_Error("\nLoad_Object_MD2 - couldn't find file %s", modelfile);
  return(0);
  } // end if

// 移动文件末尾
fseek(fp, 0, SEEK_END);

// 确定文件长度
flength = ftell(fp);

// 将.md2 文件读入缓冲区中，并对其进行分析

// 将文件指针重新指向文件开头
fseek(fp, 0, SEEK_SET);
// 分配用于存储文件数据的内存
```

```
buffer = (UCHAR *)malloc(flength+1);

// 将数据读入到缓冲区中
int bytes_read = fread(buffer, sizeof(UCHAR), flength, fp);

// 缓冲区的开头为文件头，因此用文件头指针指向它
// 以便能够对其进行分析
md2_header = (MD2_HEADER_PTR)buffer;

// 检查文件类型
if (md2_header->identifier != MD2_MAGIC_NUM ||
  md2_header->version != MD2_VERSION)
  {
  fclose(fp);
  return(0);
  } // end if

// 设置容器类的字段
obj_md2->state     = 0;          // 模型状态
obj_md2->attr      = attr;       // 模型属性
obj_md2->color     = color;      // 没有纹理时的颜色
obj_md2->num_frames   = md2_header->num_frames; // 帧数
obj_md2->num_polys    = md2_header->num_polys; // 多边形数
obj_md2->num_verts    = md2_header->num_verts; // 顶点数
obj_md2->num_textcoords = md2_header->num_textcoords; // 纹理坐标数
obj_md2->curr_frame    = 0;             // 当前帧
obj_md2->skin       = NULL;          // 指向纹理的指针
obj_md2->world_pos   = *pos;          // 在世界坐标系中的位置

// 分配用于存储网格数据的内存
// 指向多边形列表的指针
obj_md2->polys = (MD2_POLY_PTR)malloc(md2_header->num_polys*sizeof(MD2_POLY));
// 指向顶点坐标列表的指针
obj_md2->vlist = (VECTOR3D_PTR)malloc(md2_header->num_frames *
   md2_header->num_verts* sizeof(VECTOR3D));
// 指向纹理坐标列表的指针
obj_md2->tlist = (VECTOR2D_PTR)malloc(md2_header->num_textcoords *
         sizeof(VECTOR2D));

for (int tindex = 0; tindex < md2_header->num_textcoords; tindex++)
  {
  // 将纹理坐标插入到容器类中
  obj_md2->tlist[tindex].x =
    ((MD2_TEXTCOORD_PTR)(buffer+md2_header->offset_textcoords))[tindex].u;
  obj_md2->tlist[tindex].y =
    ((MD2_TEXTCOORD_PTR)(buffer+md2_header->offset_textcoords))[tindex].v;
  } // end for vindex


for (int findex = 0; findex < md2_header->num_frames; findex++)
  {

  MD2_FRAME_PTR frame_ptr =
     (MD2_FRAME_PTR)(buffer + md2_header->offset_frames +
            md2_header->framesize * findex);

  // 提取缩放因子和平移因子
  float sx = frame_ptr->scale[0],
     sy = frame_ptr->scale[1],
     sz = frame_ptr->scale[2],
     tx = frame_ptr->translate[0],
     ty = frame_ptr->translate[1],
     tz = frame_ptr->translate[2];

  for (int vindex = 0; vindex < md2_header->num_verts; vindex++)
    {
    VECTOR3D v; // 临时变量
```

```
    // 对顶点坐标进行缩放和平移
    v.x = (float)frame_ptr->vlist[vindex].v[0] * sx + tx;
    v.y = (float)frame_ptr->vlist[vindex].v[1] * sy + ty;
    v.z = (float)frame_ptr->vlist[vindex].v[2] * sz + tz;

    // 根据传入的数据进行缩放
    v.x = scale->x * v.x;
    v.y = scale->y * v.y;
    v.z = scale->z * v.z;

float temp; // 临时变量，用于坐标互换

    // 检查是否需要反转坐标
    if (vertex_flags & VERTEX_FLAGS_INVERT_X)
      v.x = -v.x;

    if (vertex_flags & VERTEX_FLAGS_INVERT_Y)
      v.y = -v.y;

    if (vertex_flags & VERTEX_FLAGS_INVERT_Z)
      v.z = -v.z;

    if (vertex_flags & VERTEX_FLAGS_SWAP_YZ)
      SWAP(v.y, v.z, temp);

    if (vertex_flags & VERTEX_FLAGS_SWAP_XZ)
      SWAP(v.x, v.z, temp);

    if (vertex_flags & VERTEX_FLAGS_SWAP_XY)
      SWAP(v.x, v.y, temp);

    // 将顶点插入到顶点列表中
    obj_md2->vlist[vindex + (findex * obj_md2->num_verts)] = v;
    } // end vindex
  } // end findex


MD2_POLY_PTR poly_ptr = (MD2_POLY_PTR)(buffer + md2_header->offset_polys);

for (int pindex = 0; pindex < md2_header->num_polys; pindex++)
  {
  // 将多边形插入容器类的多边形列表中
  if (vertex_flags & VERTEX_FLAGS_INVERT_WINDING_ORDER)
    {
    // 反转环绕顺序

    // 顶点
    obj_md2->polys[pindex].vindex[0] = poly_ptr[pindex].vindex[2];
    obj_md2->polys[pindex].vindex[1] = poly_ptr[pindex].vindex[1];
    obj_md2->polys[pindex].vindex[2] = poly_ptr[pindex].vindex[0];

    // 纹理坐标
    obj_md2->polys[pindex].tindex[0] = poly_ptr[pindex].tindex[2];
    obj_md2->polys[pindex].tindex[1] = poly_ptr[pindex].tindex[1];
    obj_md2->polys[pindex].tindex[2] = poly_ptr[pindex].tindex[0];
    } // end if
   else
    {
    // 不改变环绕顺序
    // 顶点
    obj_md2->polys[pindex].vindex[0] = poly_ptr[pindex].vindex[0];
    obj_md2->polys[pindex].vindex[1] = poly_ptr[pindex].vindex[1];
    obj_md2->polys[pindex].vindex[2] = poly_ptr[pindex].vindex[2];

    // 纹理坐标
    obj_md2->polys[pindex].tindex[0] = poly_ptr[pindex].tindex[0];
```

```
    obj_md2->polys[pindex].tindex[1] = poly_ptr[pindex].tindex[1];
    obj_md2->polys[pindex].tindex[2] = poly_ptr[pindex].tindex[2];

    } // end if

  } // end for vindex

// 关闭文件
fclose(fp);

//////////////////////////////////////////////////////////////////////////
// 从磁盘加载纹理
Load_Bitmap_File(&bitmap16bit, texturefile);

// 创建大小和位深合适的位图
obj_md2->skin = (BITMAP_IMAGE_PTR)malloc(sizeof(BITMAP_IMAGE));

// 初始化位图
Create_Bitmap(obj_md2->skin,0,0,
       bitmap16bit.bitmapinfoheader.biWidth,
       bitmap16bit.bitmapinfoheader.biHeight,
       bitmap16bit.bitmapinfoheader.biBitCount);

// 加载位图图像
Load_Image_Bitmap16(obj_md2->skin,&bitmap16bit,0,0,BITMAP_EXTRACT_MODE_ABS);

// 处理完毕，卸载位图
Unload_Bitmap_File(&bitmap16bit);

// 释放临时缓冲区
if (buffer)
  free(buffer);

// 成功返回
return(1);

} // end Load_Object_MD2
```

要调用函数 Load_Object_MD2( )，需要传递一个容器指针（该容器用于存储 MD2 模型中的数据），一个位于磁盘中的.MD2 文件的名称，缩放因子、位置和旋转角度（当前只使用缩放因子和位置），一个用于存储 16 位、256×256 纹理的.BMP 文件的名称，用于模型中多边形属性，基本颜色（没有纹理时多边形的颜色——译者注），以及参数 vertex_flags（指定是否修改环绕顺序和坐标系等）

这个.MD2 文件加载函数的原型与其他文件加载函数有些类似，读者对其中的参数应该相当熟悉。该函数首先在磁盘中查找指定的.MD2 文件，然后打开它并将其加载到一个大型缓冲区中，然后接入分析阶段：读取文件头中的信息，并根据这些信息从该.MD2 模型中提取所有重要的信息。与此同时，将所有数据以更为清晰、非压缩的方式存储到容器类中，并初始化容器类的所有字段。

vertex_flag 是最重要的参数之一，它让调用者能够修改坐标系和环绕顺序。读者可能还记得，我们使用的是左手坐标系，顶点环绕方向为顺序针，而.MD2 文件使用的是右手坐标系，顶点环绕方向为逆时针。为解决这种问题，只需互换 y 轴和 z 轴；如果不使用上述参数，这种问题解决起来将非常麻烦。加载模型并将其数据提取到 MD2_CONTAINER 中后，便可以实现动画了。为此，需要编写一个从容器中提取一帧，并将其存储到 OBJECT4DV2 中的函数。另外，还需要用于对选择的动作进行控制和执行定时（timing）的支持函数。

15.3.3 使用.MD2 文件实现动画

理解了.MD2 文件格式，将数据从压缩格式转换为更容易处理的格式后，实现动画便是小菜一碟了。完成上述功能的是加载函数和 MD2_CONTAINER 类。使用容器类来调用加载函数的方式如下：

```
// 一些工作向量
static VECTOR4D vs = {4,4,4,1};
static VECTOR4D vp = {0,0,0,1};
```

```
// 加载.MD2 模型
Load_Object_MD2(&obj_md2,   // 用于存储.md2 文件数据的容器类
    "./md2/q2mdl-tekkblade/tris.md2", //.MD2 模型的文件名
    &vs, // 缩放因子和位置
    &vp,
    NULL,
    "./md2/q2mdl-tekkblade/blade_black.bmp", //纹理的文件名
    POLY4DV2_ATTR_RGB16 |
    POLY4DV2_ATTR_SHADE_MODE_FLAT |
    POLY4DV2_ATTR_SHADE_MODE_TEXTURE,
    RGB16Bit(255,255,255),
    VERTEX_FLAGS_SWAP_YZ);     // 控制环绕顺序等
```

注意：上面使用的模型名为“Tekkaman Blade”，可在网站 Polycount 中找到。该模型很好地展示了 Polycount 网站上模型的质量，开发者为 Michael Magarnigal Mellor。

我们将文件“./md2/q2mdl-tekkblade/tris.md2”定义的.MD2 模型加载到 MD2_CONTAINER obj_md2 中，使用文件“./md2/q2mdl-tekkblade/blade_black.bmp”作为纹理。

渲染时，只需从 obj_md2 中提取指定帧的顶点列表，将其插入到 OBJECT4DV2 的顶点列表中。在渲染过程中，OBJECT4DV2 中的多变形列表、纹理坐标等不变，因为它们是间接数组，同时引用的是顶点索引而不是顶点本身。因此，可以在渲染前创建一个 OBJECT4DV2，将其除顶点列表之外的其他内容设置好。

我们将设置属性、颜色、多边形列表、纹理坐标、纹理图等，但不插入任何顶点，这被称为准备工作。渲染的准备工作就绪后，我们调用一个这样的函数，它从 obj_md2 中提取指定动画帧的顶点，将它们插入到 OBJECT4DV2 中。然后，我们将该 OBJECT4DV2 传递给渲染引擎，后者对其渲染的是什么一无所知。

1. 预处理 OBJECT4DV2

首先介绍根据 MD2_CONTAINER 的内容对 OBJECT4DV2 进行预处理的函数，该函数的代码如下（删除了一些无关的错误显示代码，以节省篇幅）：

```
int Prepare_OBJECT4DV2_For_MD2(OBJECT4DV2_PTR obj, // 指向物体的指针
      MD2_CONTAINER_PTR obj_md2) // 要从中提取帧的容器类
{
// 这个函数对 OBJECT4DV2 进行预处理，以便渲染时只需将顶点数据存储到 OBJECT4DV2 中
// 它分配所需的内存，设置字段，并执行尽可能多的预先计算
// 在每帧中，不同的只有顶点列表

// 清空物体的内容
memset(obj, 0, sizeof(OBJECT4DV2));

// 将状态设置为活动、可见
obj->state = OBJECT4DV2_STATE_ACTIVE | OBJECT4DV2_STATE_VISIBLE;

// 设置一些其他的信息
obj->num_frames  = 1;  // 总是设置为 1
obj->curr_frame  = 0;
obj->attr        = OBJECT4DV2_ATTR_SINGLE_FRAME | OBJECT4DV2_ATTR_TEXTURES;

obj->num_vertices = obj_md2->num_verts;
obj->num_polys  = obj_md2->num_polys;
obj->texture    = obj_md2->skin;

// 设置物体的位置
obj->world_pos = obj_md2->world_pos;

// 分配内存
if (!Init_OBJECT4DV2(obj,
           obj->num_vertices,
           obj->num_polys,
           obj->num_frames))
 {
 Write_Error("\n(can't allocate memory). ");
 } // end if
```

```
// 根据第 0 帧计算平均半径和最大半径
// 这不准确，在执行动作期间，物体的大小可能变化很大

// 重置，以防对应的内存中有残余值
obj->avg_radius[0] = 0;
obj->max_radius[0] = 0;

// 通过循环计算最大半径
for (int vindex = 0; vindex < obj_md2->num_verts; vindex++)
  {
  float dist_to_vertex =
     sqrt(obj_md2->vlist[vindex].x * obj_md2->vlist[vindex].x +
        obj_md2->vlist[vindex].y * obj_md2->vlist[vindex].y +
        obj_md2->vlist[vindex].z * obj_md2->vlist[vindex].z );

  // 将半径累计起来，供后面计算平均半径时使用
  obj->avg_radius[0]+=dist_to_vertex;
  // 更新最大半径
  if (dist_to_vertex > obj->max_radius[0])
    obj->max_radius[0] = dist_to_vertex;

  } // end for vertex

// 计算平均半径
obj->avg_radius[0]/=obj->num_vertices;


// 复制纹理坐标列表
for (int tindex = 0; tindex < obj_md2->num_textcoords; tindex++)
  {
  obj->tlist[tindex].x = obj_md2->tlist[tindex].x;
  obj->tlist[tindex].y = obj_md2->tlist[tindex].y;
  } // end for tindex

// 复制多边形列表
for (int pindex=0; pindex < obj_md2->num_polys; pindex++)
  {
  obj->plist[pindex].vert[0] = obj_md2->polys[pindex].vindex[0];
  obj->plist[pindex].vert[1] = obj_md2->polys[pindex].vindex[1];
  obj->plist[pindex].vert[2] = obj_md2->polys[pindex].vindex[2];

  //将多边形的顶点列表指向物体的顶点列表
  obj->plist[pindex].vlist  = obj->vlist_local;

  // 设置多边形的属性
  obj->plist[pindex].attr  = obj_md2->attr;

  // 设置多边形的颜色
  obj->plist[pindex].color  = obj_md2->color;

  // 设置多边形的纹理
  obj->plist[pindex].texture = obj_md2->skin;
  // 设置纹理坐标
  obj->plist[pindex].text[0] = obj_md2->polys[pindex].tindex[0];
  obj->plist[pindex].text[1] = obj_md2->polys[pindex].tindex[1];
  obj->plist[pindex].text[2] = obj_md2->polys[pindex].tindex[2];

  // 指出顶点有纹理坐标
  SET_BIT(obj->vlist_local[ obj->plist[pindex].vert[0] ].attr,
      VERTEX4DTV1_ATTR_TEXTURE);
  SET_BIT(obj->vlist_local[ obj->plist[pindex].vert[1] ].attr,
      VERTEX4DTV1_ATTR_TEXTURE);
  SET_BIT(obj->vlist_local[ obj->plist[pindex].vert[2] ].attr,
      VERTEX4DTV1_ATTR_TEXTURE);

  // 设置材质模式
```

```
SET_BIT(obj->plist[pindex].attr, POLY4DV2_ATTR_DISABLE_MATERIAL);

// 将多边形的状态设置为活动
obj->plist[pindex].state = POLY4DV2_STATE_ACTIVE;

//将多边形的顶点列表指向物体的顶点列表
obj->plist[pindex].vlist = obj->vlist_local;

// 设置纹理坐标列表
obj->plist[pindex].tlist = obj->tlist;

// 提取顶点索引
int vindex_0 = obj_md2->polys[pindex].vindex[0];
int vindex_1 = obj_md2->polys[pindex].vindex[1];
int vindex_2 = obj_md2->polys[pindex].vindex[2];

// 计算多边形的面法线
// 顶点按顺时针方向排列，因为 u=p0->p1，v=p0->p2，n=uxv
VECTOR4D u, v, n;

// 计算 u 和 v
u.x = obj_md2->vlist[vindex_1].x - obj_md2->vlist[vindex_0].x;
u.y = obj_md2->vlist[vindex_1].y - obj_md2->vlist[vindex_0].y;
u.z = obj_md2->vlist[vindex_1].z - obj_md2->vlist[vindex_0].z;
u.w = 1;
v.x = obj_md2->vlist[vindex_2].x - obj_md2->vlist[vindex_0].x;
v.y = obj_md2->vlist[vindex_2].y - obj_md2->vlist[vindex_0].y;
v.z = obj_md2->vlist[vindex_2].z - obj_md2->vlist[vindex_0].z;
v.w = 1;

// 计算叉积
VECTOR4D_Cross(&u, &v, &n);

// 计算法线长度，并将其存储在 nlength 中
  obj->plist[pindex].nlength = VECTOR4D_Length(&n);

  }  // end for poly

// 成功返回
return(1);

} // end Prepare_OBJECT4DV2_For_MD2
```

函数 Prepare_OBJECT4DV2_For_MD2()接受两个参数：一个 OBJECT4DV2 指针和 MD2_CONTAINER 指针，后者存储了.MD2 模型的数据，前者用于接受后者的数据。该函数对 OBJECT4DV2 进行初始化，然后将 MD2_CONTAINER 中不随不同的帧而变化的数据复制到 OBJECT4DV2 中，惟一不复制的是顶点。不要求这个函数的速度很快，因为它是在初始化阶段被调用的，因此速度并不重要。这个函数为 OBJECT4DV2 创建多边形列表、纹理列表，并执行所需的计算，如面法线的长度以及平均半径和最大半径等。

警告：所有多边形法线的长度都是根据第 0 帧计算得到的，播放动画时，始终使用预先计算得到的多边形法线长度。进行光照计算时将考虑多边形法线长度，但这几乎没有什么影响。一种更精确的方法是，存储每个多边形在每帧中的法线长度，但这意味着对于每个多边形，需要一个包含 198 个元素的法线数组，然后在提取某一帧时，复制相应的法线。这样做的渲染结果与前一种方法之间几乎没什么不同。另外，平均半径也是根据第 0 帧计算得到的。

2. 从 MD2_CONTIANER 中提取帧

对 OBJECT4DV2 进行预处理后，余下的工作很简单。只需计算指定帧的顶点在顶点列表中的位置，然后将这些顶点复制到 OBJECT4DV2 中即可，代码如下：

```
int Extract_MD2_Frame(OBJECT4DV2_PTR obj, // 指向物体的指针
   MD2_CONTAINER_PTR obj_md2) // 要从其提取帧的容器类
{
```

```
// 这个函数从 MD2 容器类中提取一帧，将其存储到 OBJECT4DV2 中
// 以便能够使用现有的库来进行变换、光照处理和渲染等，从而避免编写新的函数
// 必须承认，这种提取处理是不必要的
// 但由于程序中使用的 MD2 模型较少，渲染时间比提取时间高几个数量级
// 因此提取的影响几乎可以忽略不计
// 该函数还插入新的帧
// 如果 curr_frame 的值不是整数，函数将根据小数部分将两个关键帧混合来生成新的帧

int frame_0,
  frame_1;

// 第 1 步：判断这个帧是否需要通过插值来计算得到

float ivalue = obj_md2->curr_frame - (int)obj_md2->curr_frame;

// 检查 curr_frame 是否为整数
if (ivalue == 0.0)
  {
  // 无需通过插值来计算得到
  frame_0 = obj_md2->curr_frame;

  // 检查 curr_frame 是否溢出
  if (frame_0 >= obj_md2->num_frames)
    frame_0 = obj_md2->num_frames-1;

  // 复制当前帧的顶点列表
  // 基索引为(obj_md2->num_verts * obj_md2->curr_frame)
  int base_index = obj_md2->num_verts * frame_0;

  // 从基索引指向的顶点开始复制
  for (int vindex = 0; vindex < obj_md2->num_verts; vindex++)
   {
   // 复制顶点
   obj->vlist_local[vindex].x = obj_md2->vlist[vindex + base_index].x;
   obj->vlist_local[vindex].y = obj_md2->vlist[vindex + base_index].y;
   obj->vlist_local[vindex].z = obj_md2->vlist[vindex + base_index].z;
   obj->vlist_local[vindex].w = 1;

   // 每个顶点都由一个 3D 点和一个纹理坐标组成
   // 因此相应地设置其属性
   SET_BIT(obj->vlist_local[vindex].attr, VERTEX4DTV1_ATTR_POINT);
   SET_BIT(obj->vlist_local[vindex].attr, VERTEX4DTV1_ATTR_TEXTURE);

   } // end for vindex

  } // end if
else
  {
  // 根据 ivalue，使用第 curr_frame 帧和第 curr_frame+1 帧插值计算该帧
  frame_0 = obj_md2->curr_frame;
  frame_1 = obj_md2->curr_frame+1;

  // 检查溢出情况
  if (frame_0 >= obj_md2->num_frames)
    frame_0 = obj_md2->num_frames-1;

  // 检查溢出情况
  if (frame_1 >= obj_md2->num_frames)
    frame_1 = obj_md2->num_frames-1;

  // 计算基索引
  int base_index_0 = obj_md2->num_verts * frame_0;
  int base_index_1 = obj_md2->num_verts * frame_1;

  // 通过插值计算顶点
for (int vindex = 0; vindex < obj_md2->num_verts; vindex++)
{
```

```
// 插值计算顶点
obj->vlist_local[vindex].x =((1-ivalue)*obj_md2->vlist[vindex+base_index_0].x
                + ivalue*obj_md2->vlist[vindex+base_index_1].x);
obj->vlist_local[vindex].y =((1-ivalue)*obj_md2->vlist[vindex+base_index_0].y
                + ivalue*obj_md2->vlist[vindex+base_index_1].y);
obj->vlist_local[vindex].z =((1-ivalue)*obj_md2->vlist[vindex+base_index_0].z
                + ivalue*obj_md2->vlist[vindex+base_index_1].z);
obj->vlist_local[vindex].w =1;
   // 每个顶点都由一个 3D 点和一个纹理坐标组成,
   // 因此相应地设置属性
   SET_BIT(obj->vlist_local[vindex].attr, VERTEX4DTV1_ATTR_POINT);
   SET_BIT(obj->vlist_local[vindex].attr, VERTEX4DTV1_ATTR_TEXTURE);
} // end for vindex

  } // end if

// 成功返回
return(1);

} // end Extract_MD2_Frame
```

函数 Extract_MD2_Frame()也接受两个指针，它们分别指向接受顶点信息的 OBJECT4DV2 和提供这些信息的 MD2_CONTAINER。提取的是 MD2_CONTAINER 中的第 curr_frame 帧。然而需要注意的是，该提取函数执行关键帧插值。

在函数 Extract_MD2()中，开头有一条 if 语句：

```
// 检查 curr_frame 是否为整数
if (ivalue == 0.0)
```

该语句检测 curr_frame 是否包含小数部分。如果没有，则从顶点列表中提取 curr_frame 指定的帧的顶点。例如，如果 curr_frame 为 26，则将顶点列表中第 26 帧的顶点复制到 OBJECT4DV2 中。为此，将 num_verts（每帧的顶点数）乘以 26，将乘积作为基索引，从顶点列表中复制 num_verts 个顶点到 OBJECT4DV2 中，如图 15.12 所示。如果 curr_frame 不是整数，则需要关键帧之间进行顶点混合/插值。

在图 15.13 中，以线条画的方式绘制了步行动作的几个关键帧。要使动画看起来更为流畅，可以在第 i 和 i+1 帧之间插入其他帧，这是通过插值计算每个顶点的值/位置实现的。只要没有怪异的情况发生，且每个顶点都做线性运动，插值的结果都将是合乎情理的。这也是.MD2 文件基于的理念。虽然每个文件只有 198 帧，它们被用于 20 个不同的动作，但看起来显得内容非常丰富。每个动作平均包含 10（198/20）左右，如果游戏的速度为 60 帧/秒，则每个动作的持续时间为 10/60 = 0.16 秒（160 毫秒），

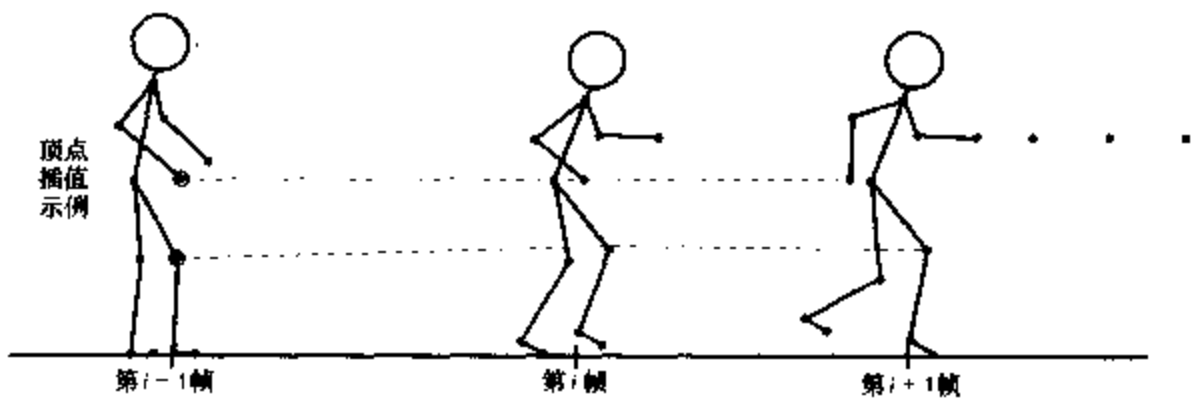


图 15.13　关键帧动画

要降低动作的速度，有两种办法：切换帧之前加入一段延迟或者插入其他帧。如果采用延迟的方法，将每帧连续播放 10 次，可将速度降低 10 倍。然而，动画看起来仍有跳跃感（jerky），因为每个动作包含的帧数太少。一种更好的解决方案是，在关键帧之间插入其他帧，具体地说是通过插值计算顶点的位置。这正是函数 Extract_MD2_Frame()中第二个代码块的功能。

3. 实现帧/顶点插值

函数 Extract_MD2_Frame()的第二个代码块比第一个代码块要复杂一些，这是因为它在两个动画帧之间进行顶点插值。根据 curr_frame 的值选择两个帧。基本上，如果 curr_frame 不是整数，假设为 5.1，则需要

将第 5 帧和第 6 帧的顶点进行混合，它们的权重分别是 90%和 10%。

给定第 0 帧的顶点 v0 和第 1 帧中的顶点 v1，根据 curr_frame 的小数部分（ivalue）插值计算新的顶点：

```
vi = v0 * (1 - ivalue) + v1 * (ivalue)
```

代码中正是这样做的。作者将 curr_frame 为整数作为特例进行处理的惟一原因是为提高速度。也就是说，如果 curr_frame 为整数，则没有必要进行插值计算，从而避免了执行一些乘法和加法运算。

curr_frame 可以是 0～197 之间的任何值，必要时，函数 Extract_MD2_Frame( )将通过插值计算出新的帧，并将其加入到 OBJECT4DV2 中。

问题的最后一部分是选择动作和帧的逻辑。

4. 添加基于状态的动画支持

现在，我们能够从磁盘中加载.MD2 文件，将其转换为更易于使用的 MD2_CONTAINER 类，从中提取帧，并将其存储到 OBJECT4DV2 中。本可以就此打住，调用渲染函数来渲染 OBJECT4DV2。然而，考虑到表 15.1 列出的动画帧，如果有一个能够对 MD2_CONAINTER 进行初始化，并控制动作播放方式（播放一次还是循环播放）的函数，那该多好。这样在游戏中，便可以根据游戏命令角色奔跑、跳跃、站着不动等，而无需使用硬编码。下面来编写完成这些任务的函数——它们非常简单。

为此，需要两样东西：一个用于表示动作的简单数据结构以及两个指定动作和更新 MD2_CONTAINER 中字段的函数。先来介绍存储动作的数据结构。动作数据结构由起始帧、终止帧、插帧频率（interpolation rate）和切换速度（animation speed）组成。

```
// 存储动作的起始帧和结束帧以及插帧频率
typedef struct MD2_ANIMATION_TYP
    {
    int  start_frame; // 起始帧和结束帧
    int  end_frame;
    float irate;    // 插帧频率
    int  anim_speed;   // 切换速度
    } MD2_ANIMATION, *MD2_ANIMATION_PTR;
```

字段 start_frame 和 end_frame 用于存储如表 15.1 所示的索引，这里有必要解释一下字段 irate 和 anim_speed。irate 是插值频率，即在两个关键帧之间应插入多少帧。例如，如果 irate 为 0.1，则需要在第 5 帧和第 6 帧之间插入如下帧：

5.0, 5.1, 5.2, 5.3 … 5.9, 6.0

在大多数情况下，将 irate 设置为 0.25 或 0.5 的效果都不错。应尽可能使用非循环小数，例如，0.33 就是一种糟糕的选择。字段 anim_speed 是一个计数器，用于在游戏运行时控制动作切换速度。将其设置为 0 时，将立刻切换到下一个动作，设置为 1 时将在播放一帧后再切换到下一个动作，依次类推。接下来需要做的是，创建一个动作结构数组，每个结构都包含如表 15.1 所示的帧索引：

```
MD2_ANIMATION md2_animations[NUM_MD2_ANIMATIONS] =
{
// 格式为：起始帧（0～197）、结束帧(0～197)、
// 插帧频率（0～1，1 表示不插帧）、
// 切换速度(0～10，切换速度从 0～10 依次降低)

{0,39,0.5,1},       // MD2_ANIM_STATE_STANDING_IDLE    0
{40,45,0.5,2},      // MD2_ANIM_STATE_RUN              1
{46,53,0.5,1},      // MD2_ANIM_STATE_ATTACK           2
{54,57,0.5,1},      // MD2_ANIM_STATE_PAIN_1           3
{58,61,0.5,1},      // MD2_ANIM_STATE_PAIN_2           4
{62,65,0.5,1},      // MD2_ANIM_STATE_PAIN_3           5
{66,71,0.5,1},      // MD2_ANIM_STATE_JUMP             6
{72,83,0.5,1},      // MD2_ANIM_STATE_FLIP             7
{84,94,0.5,1},      // MD2_ANIM_STATE_SALUTE           8
{95,111,0.5,1},     // MD2_ANIM_STATE_TAUNT            9
{112,122,0.5,1},    // MD2_ANIM_STATE_WAVE             10
{123,134,0.5,1},    // MD2_ANIM_STATE_POINT            11
{135,153,0.5,1},    // MD2_ANIM_STATE_CROUCH_STAND     12
```

```
{154,159,0.5,1},    // MD2_ANIM_STATE_CROUCH_WALK      13
{160,168,0.5,1},    // MD2_ANIM_STATE_CROUCH_ATTACK    14
{169,172,0.5,1},    // MD2_ANIM_STATE_CROUCH_PAIN      15
{173,177,0.25,0},   // MD2_ANIM_STATE_CROUCH_DEATH     16
{178,183,0.25,0},   // MD2_ANIM_STATE_DEATH_BACK       17
{184,189,0.25,0},   // MD2_ANIM_STATE_DEATH_FORWARD    18
{190,197,0.25,0},   // MD2_ANIM_STATE_DEATH_SLOW       19
};
```

在大多数情况下，作者将插帧频率设置为 0.5，将动作切换速度设置为 1。有了动作数据结构后，需要两个指定和播放动作的函数。下面是指定动作的函数：

```
int Set_Animation_MD2(MD2_CONTAINER_PTR md2_obj, // 容器类
        int anim_state,        // 播放哪个动作
        int anim_mode = MD2_ANIM_LOOP) // 播放模式，播放一次还是循环播放
{
// 这个指定要播放的动作，并相应地设置容器类
md2_obj->anim_state  = anim_state;
md2_obj->anim_counter = 0;
md2_obj->anim_speed  = md2_animations[anim_state].anim_speed;
md2_obj->anim_mode   = anim_mode;

// 设置起始帧
md2_obj->curr_frame  = md2_animations[anim_state].start_frame;

// 设置播放完毕标记
md2_obj->anim_complete = 0;
// 成功返回
return(1);

} // end Set_Animation_MD2
```

函数 Set_Animation_MD2()接受指向 MD2_CONTAINER 的指针以及动画状态和动画模式。例如，假设已经将模型加载到 MD2_CONTAINER obj_md2 中，要指定动作并以循环模式播放它，可以这样做：

```
Set_Animation_MD2(&obj_md2,
          MD2_ANIM_STATE_RUN,
          MD2_ANIM_LOOP);
```

我们还需要切换动作和修改 MD2_CONTAINER 中 curr_frame 字段，完成这种任务的函数如下：

```
int Animate_MD2(MD2_CONTAINER_PTR md2_obj) // MD2 容器类
{
// 根据状态和插值频率确定接下来要播放的帧

// 更新动作计数器
if (++md2_obj->anim_counter >= md2_obj->anim_speed)
  {
  // 重置计数器
  md2_obj->anim_counter = 0;

  // 通过加上插帧频率来计算接下来要播放的帧
  md2_obj->curr_frame+=md2_animations[md2_obj->anim_state].irate;

  // 判断动作是否播放完毕
  if (md2_obj->curr_frame > md2_animations[md2_obj->anim_state].end_frame)
   {
   // 判断是播放一次还是重复播放
   // 如果是前者，则播放最后 1 帧，如果是后者，则播放第 1 帧
   if (md2_obj->anim_mode == MD2_ANIM_LOOP)
     {
     // 播放第 1 帧
     md2_obj->curr_frame = md2_animations[md2_obj->anim_state].start_frame;
     } // end if
   else
     {
     // MD2_ANIM_SINGLE_SHOT
```

```
        md2_obj->curr_frame = md2_animations[md2_obj->anim_state].end_frame;

        // 设置播放结束标记
        md2_obj->anim_complete = 1;
        } // end else

    } // end if sequence complete

  } // end if time to animate

// 成功返回
return(1);

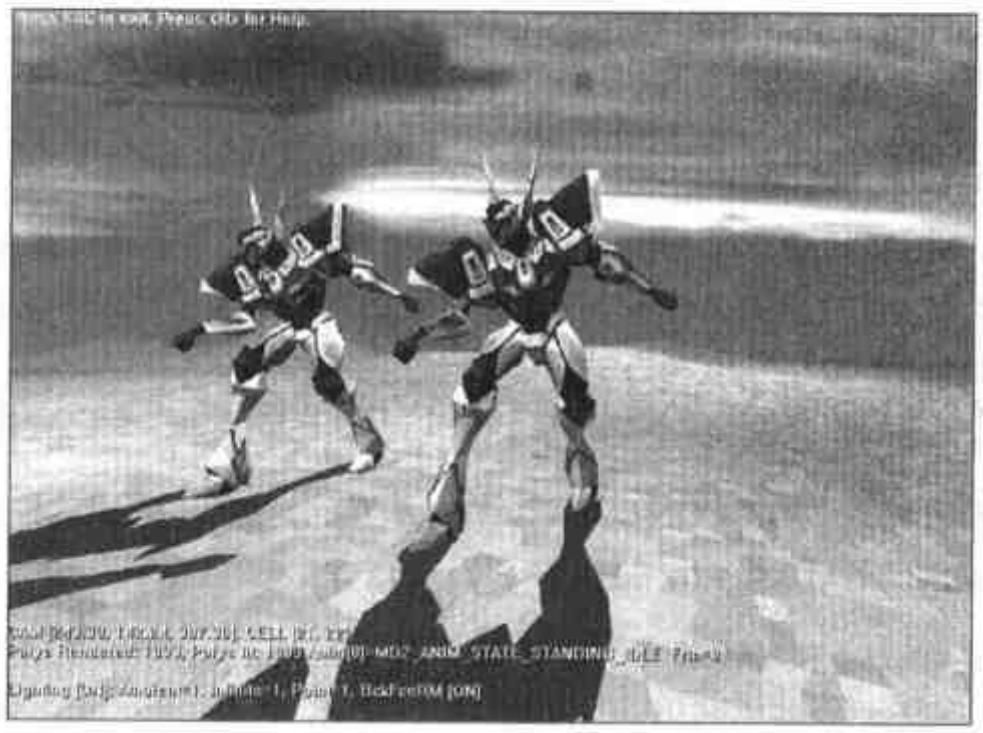
} // end Animate_MD2
```

要调用函数 Animate_MD2()，只需使用一个指向容器类的指针即可。该函数不涉及插帧，只考虑切换到当前动作的速度以及播放一次还是循环播放。

有关.MD2 文件格式就介绍到这里。现在，我们能够加载.MD2 文件，将其转换为 OBJECT4DV2，制作并渲染动画。下面来看一个演示这些技术的程序。

15.3.4 .MD2 演示程序

图 15.14 是一个加载.MD2 文件，进而播放动画的演示程序的屏幕截图。它描述了 DEMOII15_1.CPP|EXE



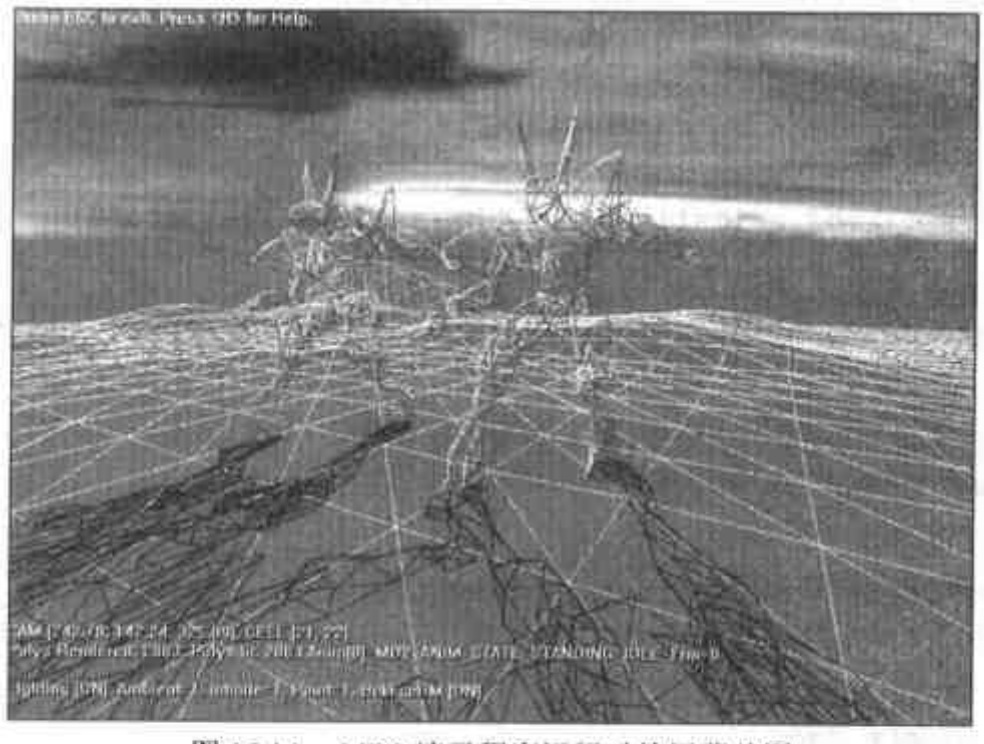


图 15.14 .MD2 演示程序运行时的屏幕截图

以线框模式和实体模型运行时的情况。该演示程序从光盘/硬盘中加载一个.MD2 文件，然后播放其中的动作。控制方法如下：

- 1——前一个动作；
- 2——下一个动作；
- 3——播放当前动作一次；
- 4——循环地播放当前动作；
- H——打开/关闭屏幕上的帮助信息；
- 箭头键——移动相机；
- Esc——退出程序。

其他控制方法请参阅屏幕上的帮助信息。如果读者不喜欢当前的定时方式，可修改数组 md2_animations[ ] 中的 irate 和 anim_speed。最后，要编译该程序，需要 DEMOII15_1.CPP|H 和所有的库文件（T3DLIB1-13.CPP|H），还需要链接 DirectX .LIB 文件。当然，在附带光盘中，有一个已经编译好的.EXE 文件。

警告：和其他演示程序一样，该程序也使用了子目录中的资源文件（.MD2 文件和纹理）。将附带光盘中的内容复制到硬盘中时，请不要修改目录结构。程序中使用的路径是相对的，因此子目录相对于.EXE 文件所在根目录的位置必须正确。

15.4　不基于角色的简单动画

到目前为止，本书一直在狂热地编写核心代码，本节将改变这种做法，只概述余下的主题，而不深入探讨。其原因在于，《Windows 游戏编程技巧》和本书前面以某种方式探讨过这些内容。尽管如此，作者还是要简要地介绍这些主题，因为很多人总是将简单问题复杂化，作者希望读者认识到，这些主题并不复杂——读者只需知道解决方法即可。下面介绍如何移动物体，以实现动画。

前面介绍了如何实现基于.MD2 的角色动画，接下来只介绍如何将物体视为点以移动它。可以使用角色动画技术和下面将介绍的思想来实现完整的动画系统，也就是说，使物体看起来像在行走或旋转，同时通过播放动画帧，使角色执行步行等动作。

15.4.1　旋转运动和平移运动

先介绍可对 3D 物体执行的两种最基本的运动：旋转和平移。要让物体边移动边转动或者固定在一个地方旋转（如炮塔），可使用旋转运动；要将物体从一个地方移到另一个地方，只能使用平移运动。

1. 旋转物体

要旋转物体，可以根据 x、y、z 旋转角度创建一个旋转变换矩阵，然后将该矩阵应用于物体。我们在很多演示程序中这样做过了，这里要做的是创建一种高级功能来驱动旋转。

另外，还需要考虑对模型的局部坐标还是变换后的坐标进行变换。读者可能还记得，所有模型都有一组局部/模型坐标，它们存储在数组 vlist[ ]中，这是模型的原始副本。对物体进行变换时，可以对其局部坐标进行变换，也可以复制这些坐标，并将变换结果存储在变换后的坐标数组 tvlist[ ]中。后一种方法的效果不错，然而，在每帧中的变换结果都将丢失，物体不能“记住”您对其执行的任何处理结果。

尽管如此，这种方法通常优于对模型坐标本身进行变换，因为对模型坐标进行变换时，模型的顶点坐标将被修改，另外由于误差的累积，经过数千或数万次变换后，模型将退化（degrade）。

因此，最好的方法是，使用一个旋转变量来存储物体的当前旋转角度，然后在每帧中，根据该变量对局部坐标进行变换，并将结果存储在数组 tvlist[ ]中，用于渲染，如图 15.15 所示。

当然，这种方法的缺点是，在每帧中都必须对模型进行变换。例如，假设加载模型时，使其主轴与系统的+z 轴平行，则需要创建一个变量（假设为 rot_y），用于存储模型绕 y 轴的旋转角度。在每帧中，需要更新变量 rot_y，使用它来创建一个绕 y 轴旋转的变换矩阵，然后据此对本地坐标进行旋转，并将结果存储

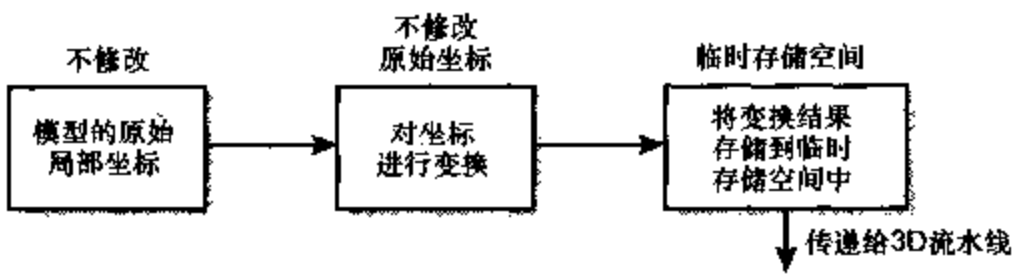


图 15.15　对局部坐标进行变换，并将结果存储到其他数组中

在变换后的顶点列表中。因此，如果模型有 1000 个顶点，则在每帧中都需要将这 1000 个顶点从模型坐标变换为世界坐标。这就是您需要付出的代价，但原始模型不会被破坏。

提示：如果为节省时间，不可能在每帧中都执行这些变换，可以直接修改模型坐标。然而，可能每经过 10 000 次变换后，都应使用原始的模型顶点来刷新模型的区部坐标，以减少由于数值误差导致的退化。

总之，作者建议尝试使用这两种方法，以了解哪种方法的效果更好。另外，花些时间来编写这样的函数是绝对值得的：命令物体每隔一段时间或一定的帧数后旋转某种角度，然后停下来。与为.MD2 系统编写动画控制代码一样，读者可能还需要编写一些辅助函数，来启动物体的旋转，让物体以某种速度在每帧中旋转。

2. 沿直线移动物体

让物体沿直线移动很容易，因为这只不过是平移而已。然而，实现这种移动的方法很多，其中一种方法是，根据速度向量 v 和物体的位置向量 P，在每帧中执行如下变换：

$$P = P + v$$

这样，物体将从起点 P 沿向量 v 向前移动。然而，如果要将物体从 P_0 移到 P_1 该如何办呢？这也很简单：只需创建一个向量方程，然后以参数化方式沿该向量平移物体，如下所示：

给定 P_0 和 P_1，将物体的世界位置设置为 P_0，然后应用如下变换：

$$P = P_0 + t * (P_1 - P_0)$$

其中 t 为[0, 1.0]。

正如读者看到的，当 t 从 0 变化到 1.0 时，位置 P 从 P_0 沿直线移到 P_1。移动速度取决于单位时间内 t 的变化量。这是一个参数化方程，图 15.16 说明了这种移动方法。

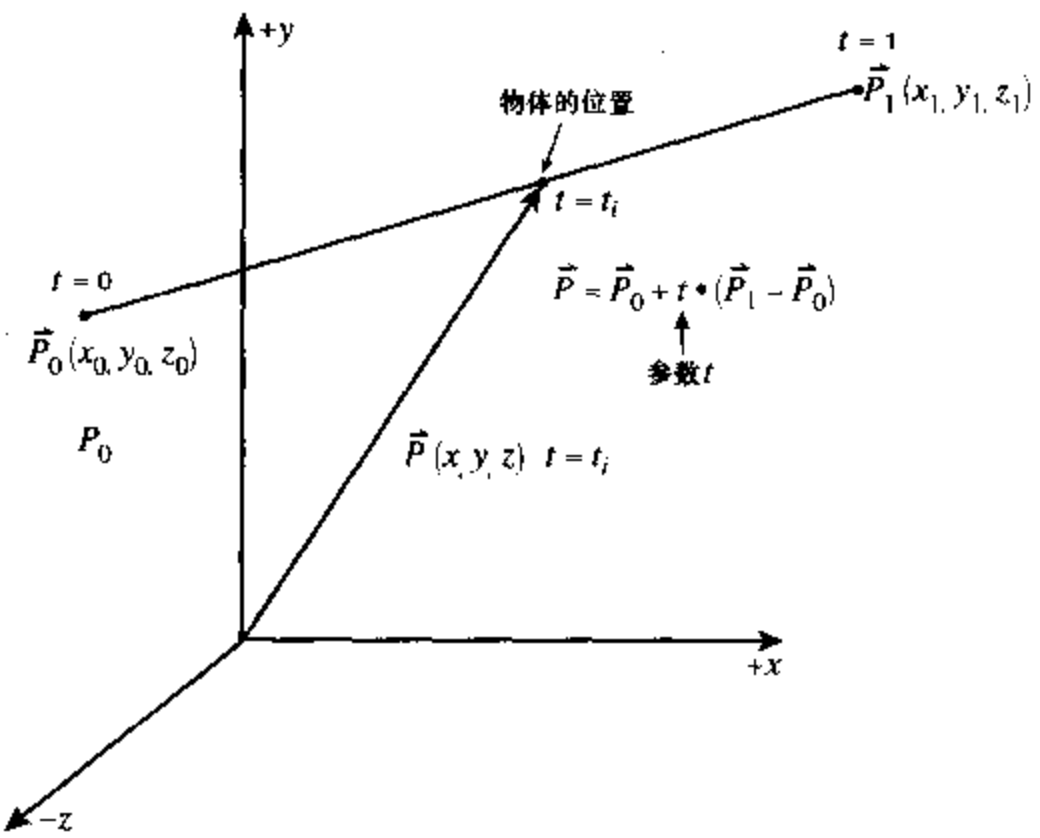


图 15.16　沿直线移动的参数化方程

15.4.2　复杂的参数化曲线移动

虽然从几何学上说，直线非常优美，但沿直线运动太单调。接下来要介绍的一种平移运动是，沿更复杂的参数化曲线移动。

注意：所有运动都可以使用基于物体的质量、加速度和环境的实时物理模型来实现，其效果看起来非常真实。然而，这里只讨论简单确定性运动，几乎不考虑各种物理量。

创建曲线运动轨迹和直线参数轨迹一样容易：只是需要使用更复杂的参数化方程。例如，要让物体沿螺旋形轨迹向上移动（如图 15.17 所示），可以使用查找表，但更好的方法是，根据运动轨迹的参数化方程来确定物体的位置。下面来推导该曲线的参数化方程。

给定位置 P，假设运动轨迹在 *x*-*z* 平面上的投影为圆或椭圆，中心为 P_0，则参数化方程如下：

```
p.x = p0.x + r1*cos(2*PI*c*t)
p.z = p0.z + r2*sin(2*PI*c*t)
p.y = p0.y + b*t;
```

其中 *r*1 和 *r*2 是分别是投影椭圆沿 *x* 和 *z* 轴的半径，*c* 为沿椭圆移动的速度，*t* 为参数，b 为垂直方向的运动速度。假设 b=c=r1=r2=1，p0=[0, 0, 0]，则参数化方程如下：

```
p.x = cos(2*PI*t)
p.z = sin(2*PI*t)
p.y = t;
```

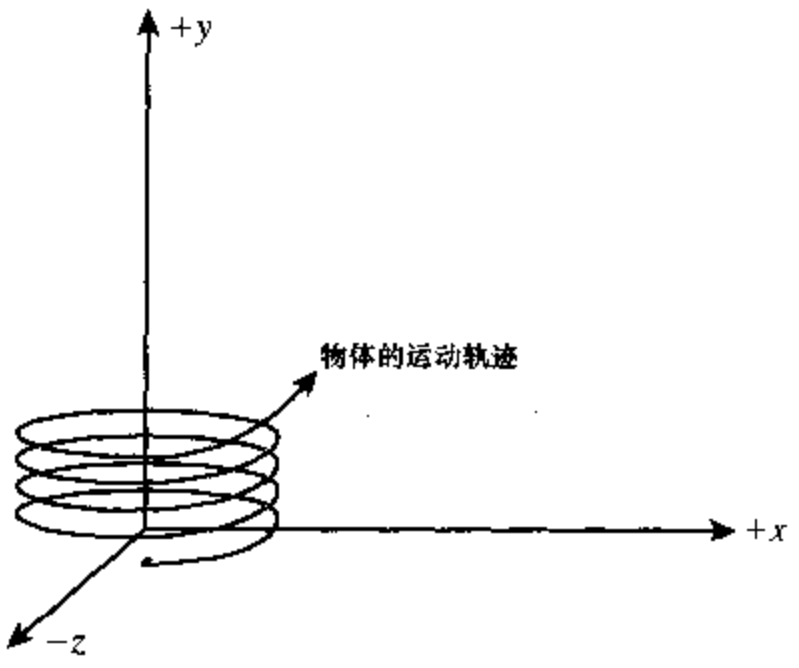


图 15.17　螺旋形运动轨迹

如果 t 的取值范围为[0, 1]，则运动轨迹为向上的螺旋线，其 *y* 坐标取值范围为[0, 1]，投影是以原点为圆心的圆。

这就是创建复杂曲线轨迹的全部内容。大多数曲线轨迹可以使用正弦/余弦曲线来创建，但还可以使用多项式、贝塞尔曲线或样条线来创建更复杂的运动轨迹。读者可能还记得，前面实现透视纹理映射时，我们使用二次曲线来近似透视曲线，进而在扫描线端点进行插值。可以使用类似的插值方法来实现运动：根据运动轨迹曲线（建议使用简单的样条线），创建用于计算 *x*、*y*、*z* 坐标的参数化方程，然后据此计算物体在该曲线上的位置，让物体沿曲线移动。当然，这需要将很多曲线段组合在一起，但这没什么大不了的。

注意：对于 3D 空间中的物体（如太空战斗机），在将其沿某种轨迹移动的同时，可能还需要让其旋转。这很容易实现：只需绕运动轨迹的切线对物体进行旋转即可。

15.4.3　使用脚本来实现运动

这里要介绍的最后一种运动技术是使用脚本，方法是创建一个小型程序，其中包含一个指示游戏角色如何运动的指令集。脚本系统的基本思想是，创建有关角色的小型脚本或行为，然后使用脚本引擎对指令进行分析，并指示角色采取相应的行动，如图 15.18 所示。

使用脚本，可以创建极其复杂的行为，让游戏角色以非常复杂的方式移动和执行动作；如果使用硬编码或运动轨迹，这些几乎是不可能实现的。另外，还可以创建类似于 C 语言的高级脚本语言，用于对角色本身进行编程。

注意：更多有关游戏脚本技术的知识，请参阅 Alex Varanese 和 John Romero 编写的 *Game Scripting Master*（Premier Press），它是市面上有关该主题的最佳图书。

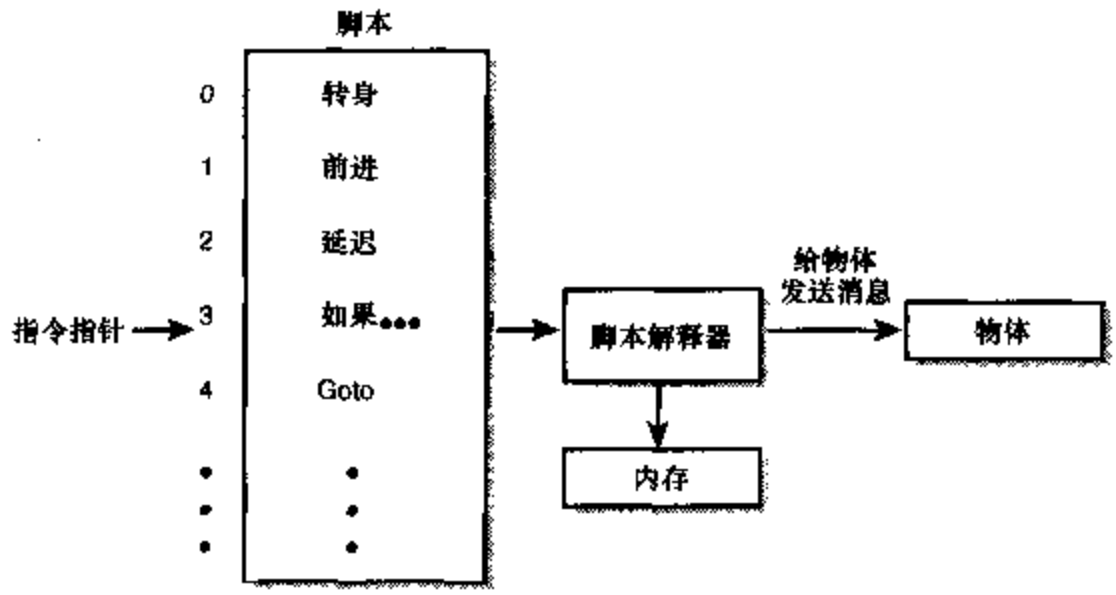


图 15.18 脚本引擎的体系结构

例如，下面来定义一种支持直线运动、绕任何轴旋转和延迟（delay）的脚本语言。每个指令都是用一条记录（数据结构）来表示的。我们将创建一个由指令数组组成的小型脚本程序，然后编写一个读取并执行指令的分析器/引擎。首先，下面是一种用于表示指令的数据结构：

```
// 一种可能的指令数据结构
typedef struct OPCODE_TYP
{
int   opcode; // 指令的操作码
float  op1;  // 浮点型操作数
VECTOR3D op2;  // 向量操作数
} OPCODE, *OPCODE_PTR;
```

每条指令都由一个操作码（opcode）和两个操作数组成。对于有些指令，需要指定操作数，有些则不需要。下面是一些操作码：

```
#define OP_MOVE          0 // 移动
#define OP_ROTATE_X      1 // 绕 x 轴旋转
#define OP_ROTATE_Y      2 // 绕 y 轴旋转
#define OP_ROTATE_Z      3 // 绕 z 轴旋转
#define OP_DELAY         4 // 延迟
#define OP_END           -1 // 退出程序
```

对于每个操作码，需要定义字段 op1 和 op2 的含义。下面是一种可能的方式：

- OP_MOVE——op2 表示移动后的位置，op1 表示速度（0 可能表示非常慢，1 可能表示非常快）。
- OP_ROTATE_*——在所有的旋转指令中，op1 表示旋转角度（单位为度），角度可以为正，也可以为负。
- OP_DELAY——op1 表示延迟帧数，必须是整数。
- OP_END——模式（pattern）引擎遇到该指令后，将停止执行模式。

这很容易，下面来实现一个简单的状态机，它这样执行程序：

```
int  instruction_ptr = 0; // 指令指针
int  fetch       = 1;   // 用于判断是否取回下一条指令
OPCODE instr;           // 用于存储下一条指令

// 下述 while 循环实际上位于函数或主循环中

while(1)
{
// 如果准备就绪，则取回下一条指令
if (fetch)
   {
   instr = pattern[instruction_ptr];
   fetch = 0;
   } // end if

// 根据操作码执行相应的操作
switch(instr.opcode)
```

```
{
case OP_MOVE:     // 移动
{
// 将物体移到 instr.op2 处

} break;
case OP_ROTATE_X:   // 绕 x 轴旋转
{
// 绕 x 轴旋转 instr.op1 度
} break;
case OP_ROTATE_Y:   // 绕 y 轴旋转
{
// 绕 y 轴旋转 instr.op1 度
} break;
case OP_ROTATE_Z:   // 绕 z 轴旋转
{
// 绕 z 轴旋转 instr.op1 度
} break;

case OP_DELAY:     // 延迟
{
// 从 instr.op1 往下倒计时
} break;

case OP_END:      // 退出程序
{
// 退出循环，终止程序等
} break;

default: break;

} // end switch
} // end while
```

代码很少，但读者应明白了其中的思想。当然，每个行为执行完成后，将把标记 fetch 重新设置为 1，这样将取回下一条指令。还需要编写一个程序，下面是沿正方形移动的程序：

```
OPCODE program[] = {
{OP_MOVE, 1, 100,0,0 },
{OP_MOVE, 1, 100,100,0 },
{OP_MOVE, 1, 0,100,0 },
{OP_MOVE, 1, 100,0,0 },
{OP_DELAY,100, 0,0,0 },
};
```

这个例子虽然非常简单，但重要的是，它演示了如何使用高级脚本（命令）来控制游戏角色的低级运动。

15.5　3D 碰撞检测

作者本打算专门辟出一章来介绍碰撞检测，但最后还是将这些篇幅用于在讨论图形学算法和渲染上，因为它们更重要，尤其是对本书而言。归根结底，碰撞检测只不过是一种几何问题，用于判断两个物体是否相交（碰撞）的算法有很多。这里要倡导的是，尽可能使其简单：在 99%的情况下，使用简单的包围球、包围长方体等的效果不逊于有些复杂的碰撞算法。下面介绍一些基本思想。

15.5.1　包围球和包围圆柱

第一种碰撞检测方法是，对于每个物体，创建一个包围它的包围球或包围圆柱，然后检测这些包围体是否发生碰撞。假设有两个物体：A 和 B，如图 15.19 所示。

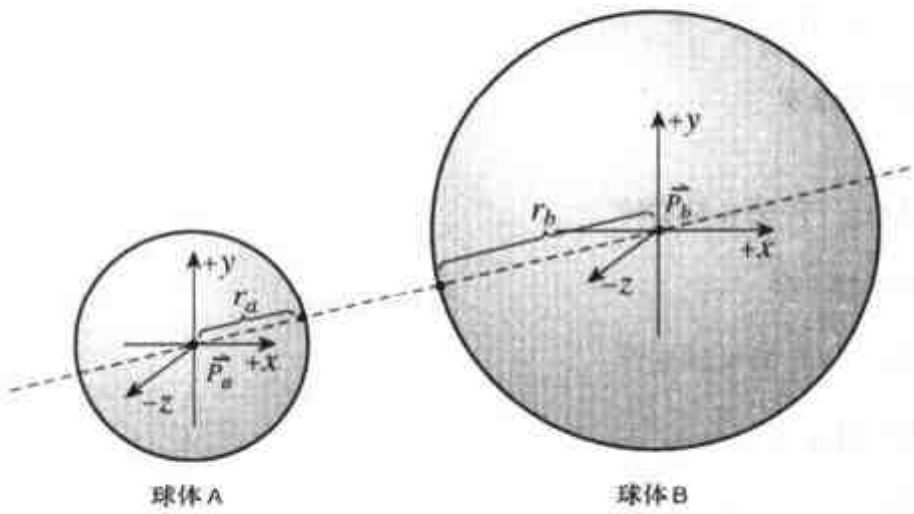


图 15.19　使用包围球来执行碰撞检测

每个物体都包含在一个包围球内，这两个包围球的半径分别是 r_a 和 r_b。要检测是否会发生碰撞，只需判断球心之间的距离是否小于 r_a+r_b，也就是说：

```
if(dist_xyz(p_a, p_b) < (r_a + r_b))
  {
  //发生了碰撞
  } //end if
```

其中 $dist_{xyz}$( )计算 p_a 和 p_b 之间的 3D 距离。

当然，如果物体本身与球形相差甚远，如长管或板子，使用包围球的效果不会太好。然而，根据作者编写快速动作游戏的经验，在这些情况下可以使用“紧缩”球体，即将包围球缩小到正常情况下的 50～70%。

另外，对于长条形物体，可以使用其他包围体，如圆柱。只要圆柱的朝向相同，便可以以类似的方法来检测碰撞，如图 15.20 所示。

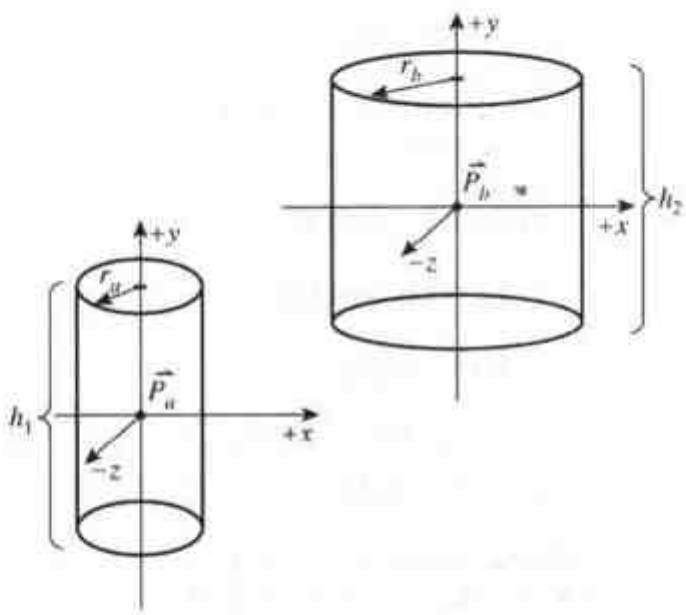


图 15.20　使用柱形包围体来进行碰撞检测

图中有两个圆柱体：A 和 B，它们的半径分别是 r_a 和 r_b。然而，要检测碰撞，需要执行两种测试：在 y 轴上是否重叠（就这个例子而言）；在半径方向上是否重叠。伪代码如下：

```
if (dist_xyz(p_a, p_b) < (r_a + r_b) &&
   (fabs(p_a.y - p_b.y) < (h_1 + h_2) / 2))
   {
   //发生了碰撞
   } //end if
```

其中 $dist_{xz}$( )计算 p_a 和 p_b 在 x-z 平面内的 2D 距离。

当然，还可以使用与问题更相符的其他包围体，如与坐标轴或物体轴平行的包围长方体。使用不同的包围体时，碰撞检测的方法是类似的——通过一系列几何测试来判断物体是否重叠。

注意：有一些很不错的有关碰撞检测几何学的图书，如 Philip J. Schneider 和 David H. Eberly 编写的 *Geometric Tools for Computer Graphics*（Morgan Kaufmann Press）以及 *Graphics and Game Programming Gems* 系列图书。

15.5.2　使用数据结构来提高碰撞检测的速度

有关碰撞检测，最后要介绍的内容与优化相关。碰撞检测技术像数学公式：我们都知道，很多书介绍，每个人都使用。实时游戏和理论碰撞检测算法之间的区别在于速度。

例如，假设游戏世界中有 1000 个物体（物体、子弹等）——在第一人称射击游戏和太空游戏中，这种情况很容易出现。要对这 1000 个物体进行碰撞检测，需要执行 1000×1000 = 1 000 000 次碰撞检测计算。如果所有物体都是武器，游戏空间中将充斥着粒子，物体总数高达 10 000 个，将需要执行 10 000×10 000 = 100 000 000 次碰撞检测计算。显然，在这种情况下，提高直线交点计算函数和包围体碰撞检测函数的速度

于事无补。作者想说的是，必须从不同的角度来解决碰撞检测问题，即使用空间划分技术。

前面讨论空间划分时介绍过 BSP 树和八叉树，同时指出过，这些数据结构不但可用于图形渲染，还可用于碰撞检测，现在读者知道其原因了吧。使用这些数据结构，可以将物体划分到不同的单元格或区域中，使得每个区域中只有两三百个甚至更少的物体，从而减少了碰撞检测的计算量。

例如，以包含 10 000 个物体的游戏空间为例，假设经过空间划分后，每个单元格中最多有 50 个物体。因此，只需处理 10 000/50 = 200 个单元格，在每个单元格中，需要执行 50×50 次碰撞检测，总共为 200×2500 = 500 000 次碰撞检测。

相对于 100 000 000 次，这少得多。换句话说，通过使用空间划分技术，将需要执行 100 000 000 次碰撞检测的问题简化为只需 500 000 次，计算量为原来的.05%（500 000/100 000 000）。

这种跳出条条框框进行思考的方式，正是进行碰撞检测时所需要的。不要将重点放在检测包围长方形或多边形相交的算法上，而应从通用碰撞检测算法着手，思考解决问题的办法。

15.5.3　地形跟踪技术

最后要讨论的一个主题让作者遇到了很多问题。编写使用地形的游戏时，必须让坦克、游戏角色等紧贴地形。在本书的大部分地形演示程序中，我们使用了一个非常简单（而有效）的物理模型，它计算移动物体的中心位于哪个地形分片的上方，确定该分片的顶点的高度，然后根据物体和地形的高度差抬高或降低物体。

这样做的效果不错，只需要 10 行代码。然而，问题是，物体并没有紧贴地面。换句话说，如果您观察物体，将发现它被正确地抬高和降低，但并没有紧贴地面。下面介绍如何解决这种问题。

图 15.21 显示了一个需要紧贴的地形分片。一种可能的地形跟踪算法的步骤如下：

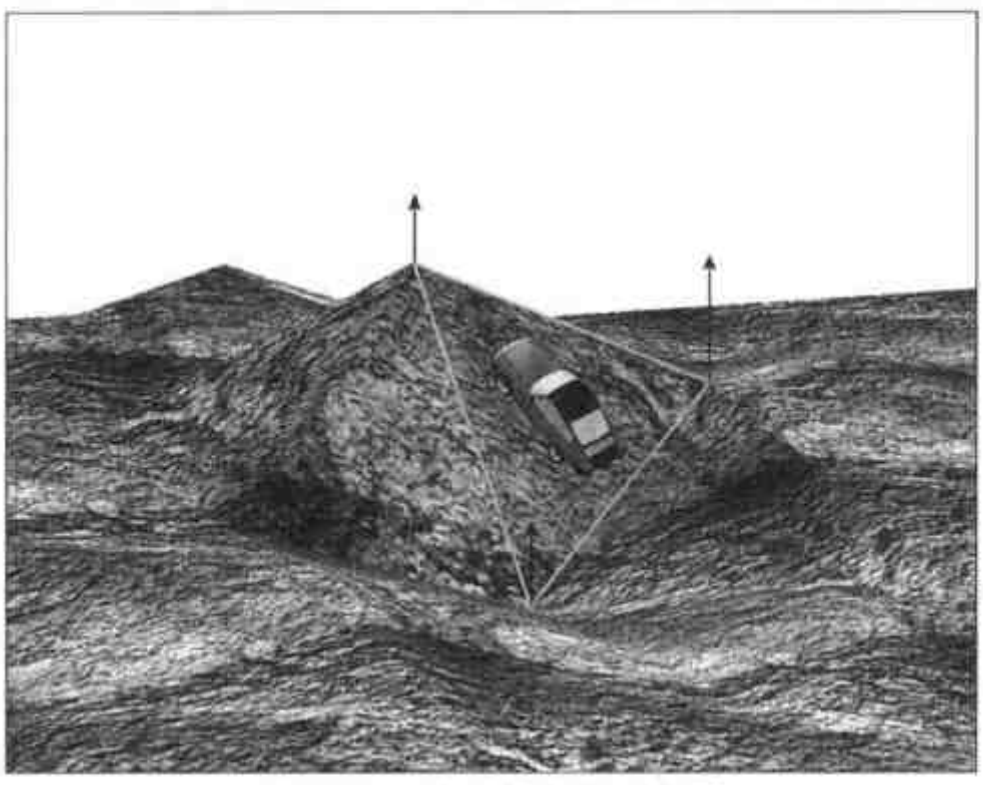


图 15.21　要解决的地形跟踪问题

1. 计算当前地形分片的顶点的平均高度，将其作为物体的高度；如果要提高准确度，应计算从物体中心出发的射线与当前地形分片的交点。当然，如果要绝对准确，应计算从轮胎或物体接触点（而不是物体中心）出发的射线与当前地形分片的交点。然后，将物体的高度设置为交点的高度，从而相应地抬高或降低物体。

2. 另一个需要考虑的因素是物体的旋转方向。有两种根据物体的接触点来解决这种问题的方法。然而，一种简捷方法是：计算当前地形分片的法线向量，然后对物体进行旋转，使其法线向量与地形分片的法线向量平行。为此，需要计算物体法线和地形分片法线之间的交角，计算方法有很多，其中之一是使用点积。计算出需要旋转的角度后，必须据此创建一个旋转矩阵，并使用它对物体进行变换，使物体的法线与地形分片的法线平行。

由于篇幅有限，这些实现细节留给读者去完成，这里只介绍一下大概步骤：首先将物体平移到正确的高度；然后对物体进行旋转，使其法线与当前地形分片的法线平行。

15.6 总　　结

本章介绍了 3D 角色模型动画以及如何移动物体。就运动和一般意义的动画而言，本书前面的章节做了大量的介绍，但有必要梳理一下，虽然这些内容很基础。最后，本章介绍了一些基本的 3D 碰撞检测技术，讨论了一些用于高级系统的理念。

下一章是本书的最后一章，将介绍一些适用于我们的引擎的优化技术，同时讨论奔腾 III 以上处理器支持的 SIMD（单指令多数据）指令。

第 16 章　优 化 技 术

id Software 的 John Carmack 说，“光速令人讨厌”，这是本章风格的真实写照：没有最快，只有更快。事实上，光速不过是海市蜃楼——在银河系中，不可能以光速运动。作者宁愿将其视为速度极限，当某个物种进化到能够突破光速的极限时，它们很可能足够聪明，不会毁灭银河系。

本章介绍一些优化技术，它们远不如突破光速极限那么复杂。这里不打算介绍低级优化技术，因为它们实现起来太耗时，而是将重点放在利用新的技术和工具来帮助完成游戏开发中优化阶段的工作。另外，在本章的最后，将介绍如何从 Intel 获得价值 250 美金的免费图书。本章将讨论的主题如下：

- 优化技术简介；
- 使用 Visual C++和 Intel VTune 剖析代码；
- 使用 Intel C++编译器；
- 单指令多数据编程初步；
- 通用优化技巧。

16.1　优化技术简介

撰写本章之前，作者做了深入的思考和研究，最终决定推翻原来的计划，从一个全新的角度撰写本章。最初，作者打算介绍一些标准内容，讨论如何对循环和代码进行优化。然而，这样的内容作者在以前的图书中介绍过，同时与稍后将讨论的内容相比，作者对这些信息是否更重要表示怀疑。作者想介绍一些真正称得上“技巧”（非常识性）的知识，让读者阅读本章后能够去尝试一些新东西。有鉴于此，作者决定将目光从具体细节转移到技术上，并自问，在优化代码时，我使用了哪些工具呢？随后，作者对众多游戏程序员进行了调查，结果令人惊讶而担忧——他们从未使用过这些工具/技术，甚至从未听说过它们。作者终于找到了很多人都不了解的东西，这些很适合在本章进行介绍。

下面提出几个问题。如果读者对其中任何一个问题做出了否定回答，则肯定会从本章受益。

- 问题 1：实际使用过 Microsoft Visual C++内置的剖析工具吗？
- 问题 2：实际使用过 Intel 的优化工具 VTune 吗？
- 问题 3：安装并使用过 Intel 提供的针对 Microsoft Visual C++的优化 C++编译器插件吗？
- 问题 4：编写过 SSE 或 SSE2 代码吗？它们分别是用于奔腾 III 和奔腾 4 的流式单指令多数据扩展（Streaming Single Instruction Extensions）第 1 版和第 2 版的简称。
- 问题 5：阅读过 Intel 奔腾 III/4 体系结构和软件开发手册吗？该手册可通过网络免费获得，Intel 还提供免费邮寄服务。

通过对大约 20 名游戏开发人员的调查发现，90%的被调查者对上述全部问题都做出了否定回答。这在作者的意料之中，但更为令人惊震的是，50%的专业游戏程序员也对大部分问题做出了否定回答。这意味着不但大部分游戏开发人员没有使用真正的优化工具，而且他们以缓慢地模式运行奔腾 III/4 处理器，没有充分利用 SIMD 和并行执行单元提供的超级标量体系结构（scalar architecture）。Darth Vader 说过，“您的不信教让我惊震”。有鉴于此，

本章将向读者表明，尝试使用这些工具和技术是多么容易——随后，一个全新的世界将展现在您面前。

当然，本章最后将给读者施一些魔法，兑现书名中所说的技巧。

16.2 使用 Microsoft Visual C++和 Intel VTune 剖析代码

在以前，可以在调用函数之前调用一个定时函数，通过调用数千次来计算函数的执行时间。或者，您可能这样做：对程序进行优化后，重新编译并运行它，尝试性地对程序进行测试。当然，这些方法现在仍然可行，对于小型程序来说效果也不错。然而，在使用先进处理器和优化编译器，且软件系统中包含系统调用、.DLL、API 和程序员代码的情况下，使用这种方法来确定从哪里着手进行优化将非常繁琐，且几乎是不可能的。

例如，编写光栅化应用程序时，您可能认为，大部分工作是在光栅化函数的内循环中完成的，因此对其进行了优化。然而，您可能没有认识到，在光栅化函数之前，调用整字节复制的 C 运行阶段函数 memcpy() 可以将速度提高 4 倍。问题在于，如果您不知道速度很慢的地方，怎么可能会对其进行优化呢。因此，我们需要一些工具（尤其是可视化工具），它们能够以表格和图形的方式指出函数的处理时间、内存使用情况和其他相关的因素，这样，您一眼就能看出整个程序的“轮廓”，准确地知道应将时间花在哪些地方。这就是程序剖析（profiling）的主旨。

当前，用于对程序进行剖析的工具有很多，这里只介绍其中的两个，其中一个读者可能有了，而另一个是免费的（至少演示版本是免费的）。

16.2.1 使用 Visual C++进行剖析

Microsoft Visual C++专业版和企业版都有内置的剖析工具。如果读者使用的是教育版（标准版），将不能使用这些工具（.NET 也有类似的剖析功能）。在 Visual C++进行剖析非常简单，它能够立刻提供一些不错的结果，让您能够进行优化分析，找出亟需优化的地方。下面将以 DEMOII16_1.CPP|EXE 为例，对其进行剖析。这基本上是前一章的角色动画演示程序，不同的是，它运行在一个窗口中，这样我们将能够移动窗口，同时如果该程序崩溃，我们仍能够获得对计算机的控制权。

要编译 DEMOII16_1.CPP|EXE，需要创建一个工程或使用已有的工程。现在，读者应该对此得心应手。当然，像前一章一样，需要在工程中包含 T3DLIB1-13.CPP 以及 DirectX .LIB 文件。编译并执行它时，读者将看到一个在窗口中运行的角色动画演示程序，如图 16.1 中的屏幕截图所示。

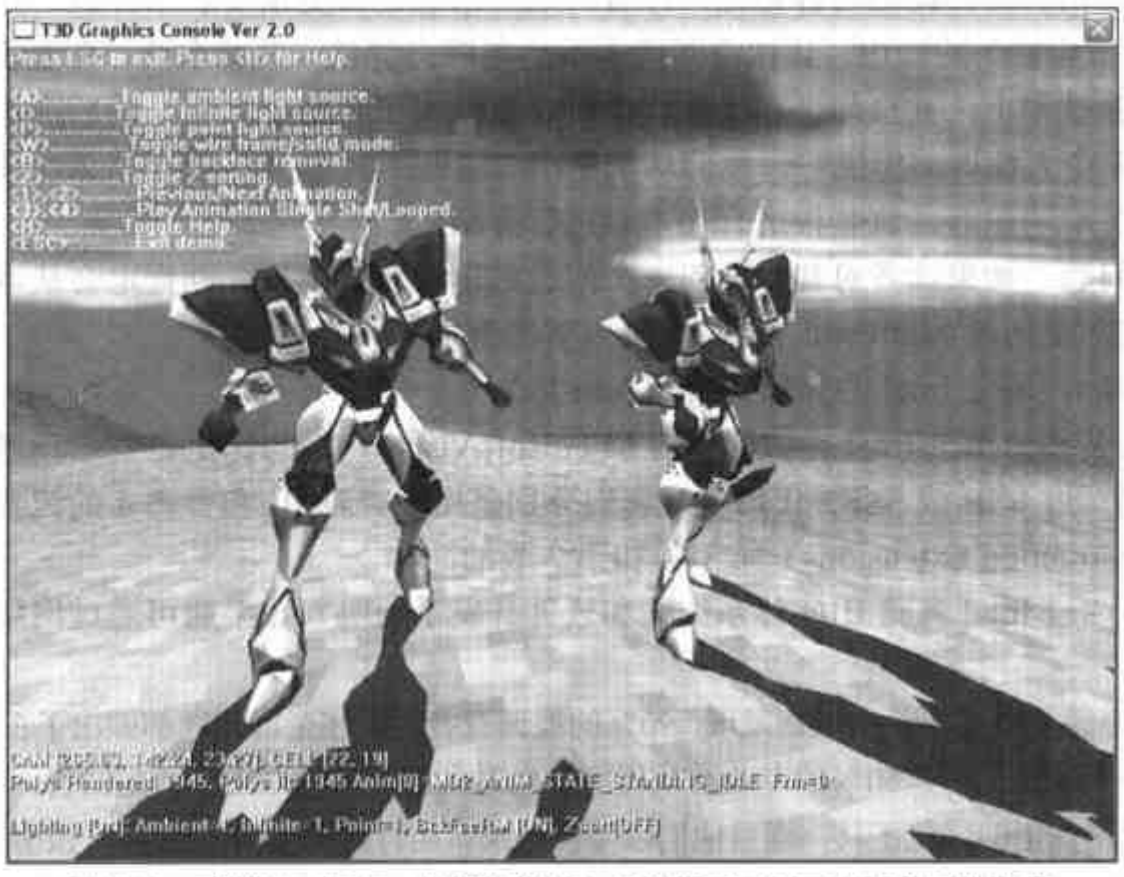


图 16.1 在窗口中运行（以简化剖析）的角色动画演示程序的屏幕截图

有了能够通过编译的演示程序后，下面来看看如何设置编译器，以便对其进行剖析。具体步骤如下：

1．从 Visual C++的主菜单栏中，选择菜单 Project/Settings。在 Project Settings 对话框中，单击选项卡 Link，然后在下拉式列表框 Category 中选择 General，如图 16.2 所示。然后，选中复选框 Enable Profiling 和 Generate Debug Info，在单击 OK 按钮，关闭该对话框。

2．现在必须使用新设置重建应用程序，为此选择菜单 Build/Rebuild All，以便在启用剖析功能的情况下重建应用程序。

3．现在可以进行剖析了，是不是很容易？要进行剖析，可选择菜单 Build/Profile，应用程序将打开一个小型对话框，让您选择剖析选项，如图 16.3 所示。选中单选按钮 Function Timing，然后单击 OK 按钮。这样应用程序将以非常慢的速度启动并运行，这是因为运行时需要进行采样。

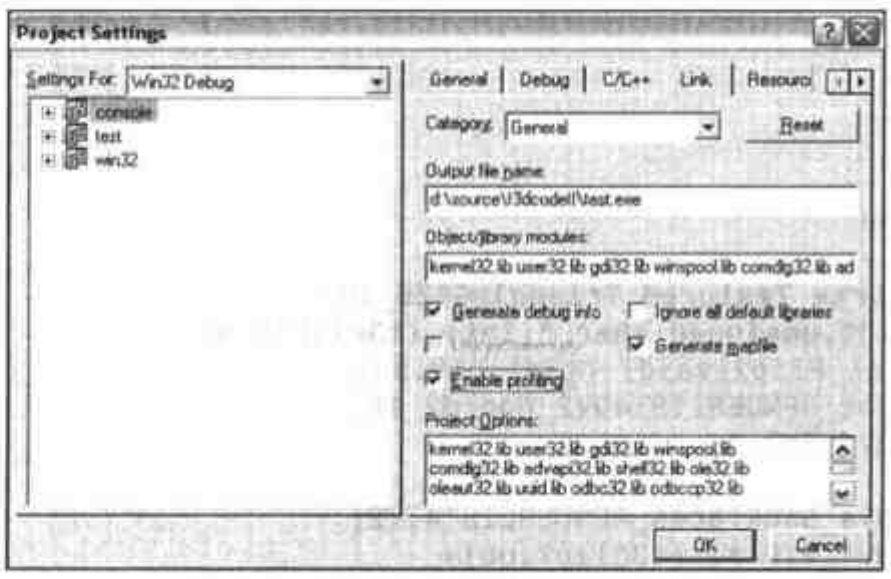


图 16.2　在 Visual C++中启用剖析功能



图 16.3　在剖析对话框中选择选项

4．让应用程序以分析方式运行 30～60 秒，然后单击窗口上的“关闭”图标或按 Esc 键，退出该程序。

5．在工作空间视图中，有一个名为 Profile 的选项卡。在大多数情况下，编译器将自动切换到该选项卡，如果没有，请使用鼠标单击它，如图 16.4 所示。然后调整窗口的大小，以便能够看到更多的信息，如图 16.5 所示。下面来看看这些信息。



图 16.4　在 Visual C++中选择剖析输出窗口

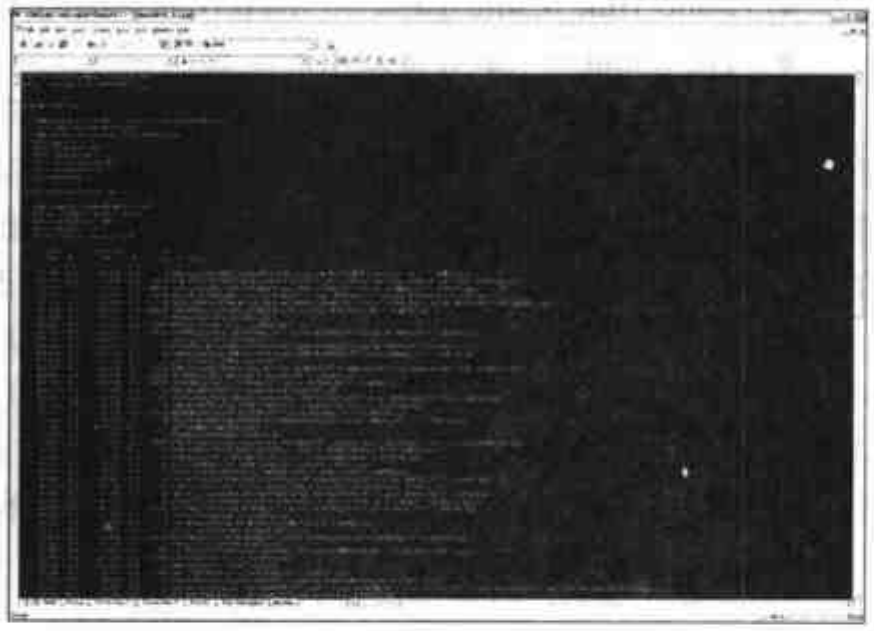


图 16.5　展开剖析视图以分析其中的数据

16.2.2　分析剖析数据

以 Function Timing 或 Function Coverage 模式运行剖析器后，将看到剖析器输出窗口，其中的内容与下面类似：

```
Profile:Function tining , sorted by time
Date:    Fri Jan 03 18:57:23 2003

Program Statistics
------------------
```

```
   Command line at 2003 Jan 03 18:55: "D:\source\t3dcodeII\test"
   Total time: 12949.659 millisecond
   Time outside of functions: 3.581 millisecond
   Call depth: 6
   Total functions: 574
   Total hits: 12775605
   Function coverage: 20.6%
   Overhead Calculated 5
   Overhead Average 5

Module Statistics for test.exe
------------------------------
   Time in module: 12946.078 millisecond
   Percent of time in module: 100.0%
   Functions in module: 574
   Hits in module: 12775605
   Module function coverage: 20.6%

      Func     Func+Child      Hit
      Time  %     Time   %   Count Function
-------------------------------------------------------------
   2751.604 21.3   2751.604 21.3  164604 Draw_Textured_TriangleGSZB_16(
struct POLYF4DV2_TYP *,unsigned char *,int,unsigned char *,int) (t3dlib10.obj)
   1115.720  8.6   1115.720  8.6   158 DDraw_Flip2(void) (t3dlib10.obj)
   1041.832  8.0   1484.683 11.5   316 Light_RENDERLIST4DV2_World2_16(
struct RENDERLIST4DV2_TYP *,struct CAM4DV1_TYP *,
struct LIGHTV2_TYP *,int) (t3dlib8.obj)
   653.307  5.0   1334.972 10.3   316 Remove_Backfaces_RENDERLIST4DV2(
struct RENDERLIST4DV2_TYP *,struct CAM4DV1_TYP *) (t3dlib7.obj)
   631.246  4.9   631.246  4.9  962220 Insert_POLY4DV2_RENDERLIST4DV2(
struct RENDERLIST4DV2_TYP *,struct POLY4DV2_TYP *) (t3dlib7.obj)
```

开头为常规信息，我们感兴趣的是下述信息：

- Func Time——函数本身的执行时间。
- Func Time+Child Time——函数及其调用的函数的总执行时间。
- Hit Count——函数被调用的次数。

虽然这些信息不是太充分，但绝对可以据此着手进行优化。如果不使用工具，将需要做很多工作才能获得这些信息。例如，来看剖析数据的第一行（前面以粗体显示的内容），这是对多边形光栅化函数的调用，其计算周期在整个应用程序占 21%。如果能够将该函数的速度提高两倍，将可以极大地提高整个应用程序的性能，因为该函数占用的时间将从 21%降低到 10.5%。

总之，对代码进行剖析是一种强大的手段，可以确定程序中哪些函数占用的时间最多，进而将优化重点放在这些函数上。虽然您可能认为光照函数的速度非常慢，但通过剖析您可能发现，其实大部分时间花在清除 *z* 缓存上（只有几行代码），这样您将把优化重点放在这几行代码上，在提高性能方面比对 500～1000 行光照处理代码进行优化的效果好得多。因此，选择攻击点至关重要。

16.2.3 使用 VTune 进行优化

每个人都听说过 VTune，但谁真正使用过它呢？那些需要强大功能的人。Intel 的 VTune 无疑是作者用过的最酷的工具之一，但与其强大功能相伴的是复杂性。Intel 做了很大的努力，使该工具对初学者来说很容易使用，任何人只需花几分钟安装并启动它，便可以立刻从中受益。

VTune 是 Intel 最重要的软件优化工具，让您能够查看、分析和优化任何程序。使用它，可以进行各种各样的分析：函数计时、内存使用情况、高速缓存访问，这个清单可以不断地列下去。当然，有关这个主题，足够编写一整本图书，这里不打算这样做，而只安装并尝试使用它。

1. 安装 VTune

首先，需要从 Intel 的网站上下载最新的 VTune 版本，下载地址如下：

http://developer.intel.com/software/products/VTune/

在该网页中找到链接 Free Evaluation Download，并下载该软件。请根据您使用的编译器，选择 Visual C++或.NET 版本。下载该应用程序后，启用安装程序，将该应用程序的默认版本安装到计算机中。安装非常简单，安装完毕后可能需要重新启动计算机。

2．为剖析做准备

安装 VTune 后，便可以进行剖析了。然后，启动 VTune 之前，先准备好要测试的应用程序。这里仍使用 DEMOII16_1.CPP|EXE，因此先启动 Visual C++，并加载该应用程序的工程，为编译做好准备。首先，需要对编译器做一些设置，步骤如下：

1．由于要对运行阶段代码进行剖析，因此最好将编译器设置为发行（release）模式。为此，可选择菜单 Build/Set Active Configuration，然后选择应用程序的发行版本，如图 16.6 所示。

2．接下来需要确保所有符号对 VTune 来说都可用，以便它能够显示“代码视图”，为此，需要让编译器能够浏览所有的数据库信息。方法是，选择菜单 Project/Settings，然后单击选项卡 C++，并在下拉式列表框 Category 中选择 General，然后选中复选框 General Browse Info，并在下拉式列表框 Debug Info 中选择 Program Database。这样，剖析器将能够看到所有的符号和代码。图 16.7 是完成上述设置后的对话框。不要关闭该对话框——我们还没有设置好。

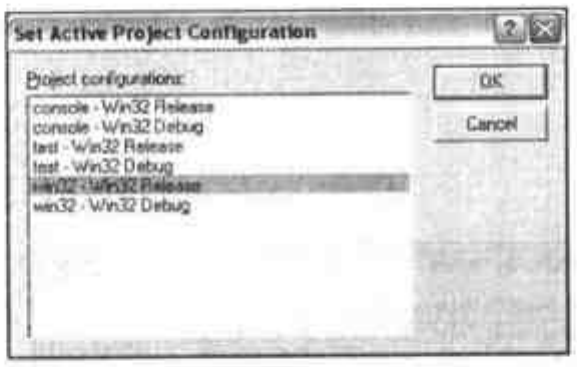


图 16.6　选择应用程序的发行版本

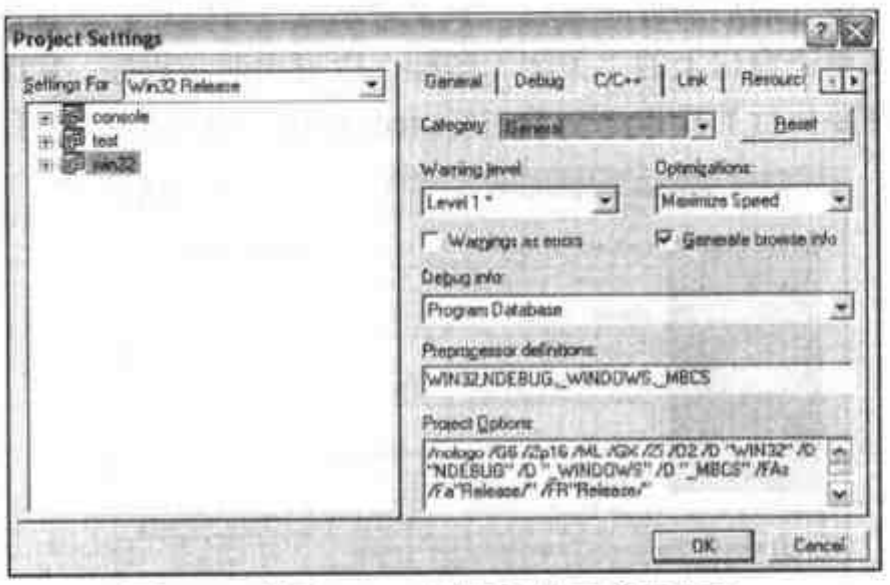


图 16.7　确保 VTune 能够看到符号和源代码

3．单击选项卡 Link，并在下拉式列表框中选择 Category。然后选中复选框 Generate Debug Info，如果您使用的是 Visual C++专业版或企业版，可能还需要选中复选框 Enable Profiling，以便能够对剖析结果进行比较。完成这些设置后，对话框如图 16.8 所示。

4．现在可以重建应用程序了，为此选择菜单 Build/Rebuild 以重建整个应用程序。

仅此而已，现在您有了最终的.EXE 文件了。

3．运行 VTune

编译测试应用程序 DEMOII16_1.CPP 并将 Visual C++窗口最小化后，便可以启动 VTune，以完成余下的工作。在您计算机桌面上，有一个 VTune 快捷方式，双击它可启动 VTune；也可以选择“开始/程序/Intel VTune Performance Analyzer”来启动。无论采取哪种方式，都将出现如图 16.9 所示的对话框。

该对话框是一个向导，让您能够创建或加载剖析工程。现在，单击按钮 Quick Performance Analysis Wizard，以创建一个小型工程。具体步骤如下：

提示：该对话框的最下面有一个名为 View Getting Started Tutorial 按钮。强烈建议读者花 20-30 分钟仔细阅读该教程——作者必须这样做，只有读者也这样做才公平！

1．单击按钮 Quick Performance Analysis Wizard 后，将出现另一个对话框，如图 16.10 所示。该对话框让您能够选择要剖析的程序，设置运行阶段环境。我们要剖析的是 DEMOII16_1.EXE，因此单击浏览按钮“...”，打开一个文件查找对话框，然后找到并选择该文件。结果如图 16.10 所示。

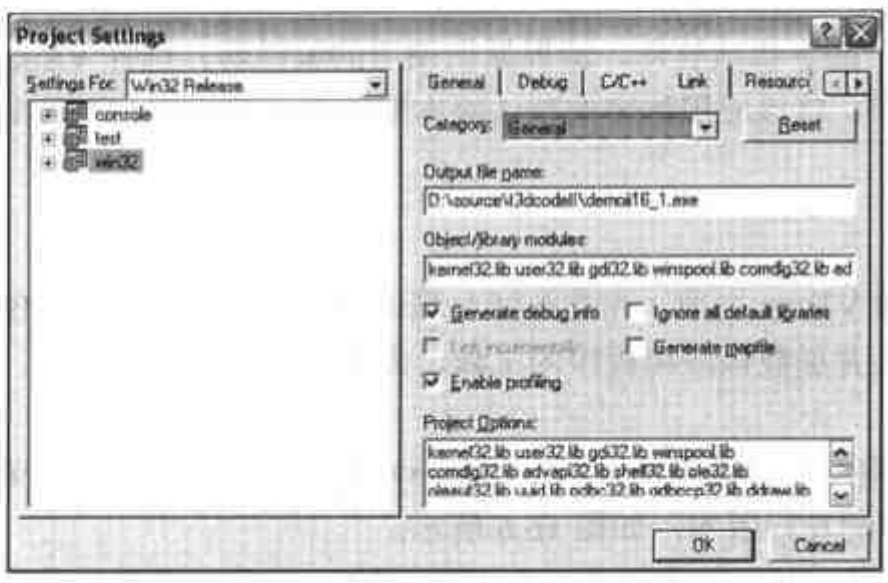


图 16.8　启用更多的调试和剖析选项



图 16.9　启动 VTune

2．接下来需要指定要执行什么样的剖析。为此，选中复选框 Collect Sampling with Counter Monitor Data 和 Collect Call Graph Data。前者表示进行计时，后者表示要生成完整的调用图，其中包含所有的函数调用树。

3．单击按钮 Go，开始进行剖析。应用程序将启动和停止多次，在最后一遍，程序运行得非常慢。这是在进行最后的计时分析，如果您愿意，可以让它一直运行下去。作者建议在程序运行 1～2 分钟后按 Esc 键或单击窗口上的“关闭”图标，以退出演示程序。VTune 将开始工作，最后出现一个小型对话框，让您选择要执行的操作——单击 Cancel 按钮，将其关闭即可。

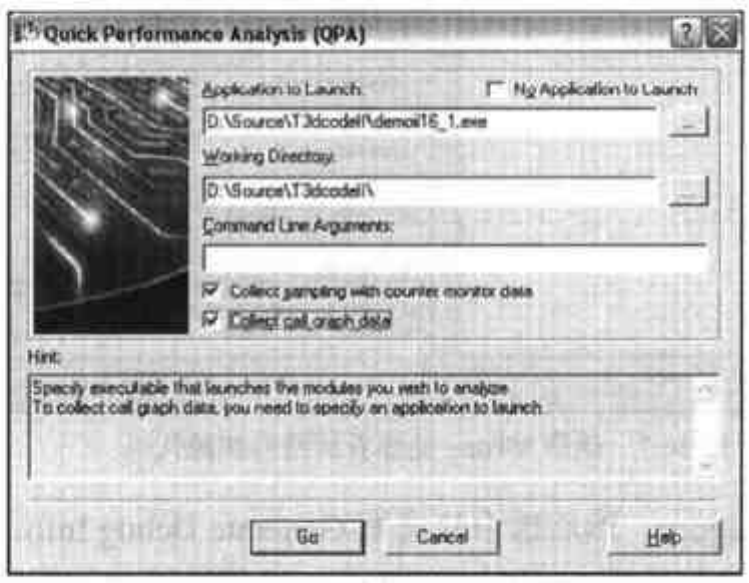


图 16.10　选择要剖析的应用程序

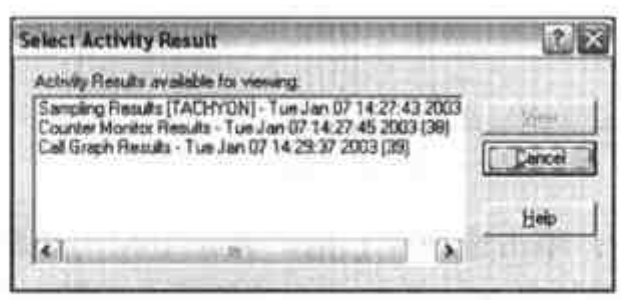


图 16.11　操作选择对话框

至此，您有了完整的剖析数据，下面简要地介绍一下其中的重要信息。

4．查看 VTune 的数据

要全面了解 VTune 支持的各种神奇的分析方法，读者必须阅读 VTune 教程。这里只介绍一些重要的方面，为读者起步做准备。

执行剖析后，出现的第一个视图由 VTune 的 GUI 和 4 个窗口组成（如图 16.12 所示）：左边的工程窗格、两个垂直窗格（数据窗格和图示窗格）和最底下的小型输出窗格。这个视图只是列出了系统级的计时信息。双击图示窗格中的任何一行，数据窗格中相应的数据将呈高亮显示。我们对这些数据不太感兴趣，下面来看看 Clockticks 视图。

（1）查看 Clockticks 视图

Clockticks 视图列出了系统中运行的每个应用程序/进程占用的系统资源。要进入该视图，可双击最左边的工程窗格中 Run 0 下面的 Clockticks 项，如图 16.13 所示。

另外，图中还有一个柱形统计图，说明了分析期间正在运行的所有进行占用的处理器时间。如果仔细查看，读者将发现应用程序 DEMOII16_1.EXE 位于图中的最下面。双击左边的文本，将打开一个有关该应用程序的详细视图，如图 16.14 所示。

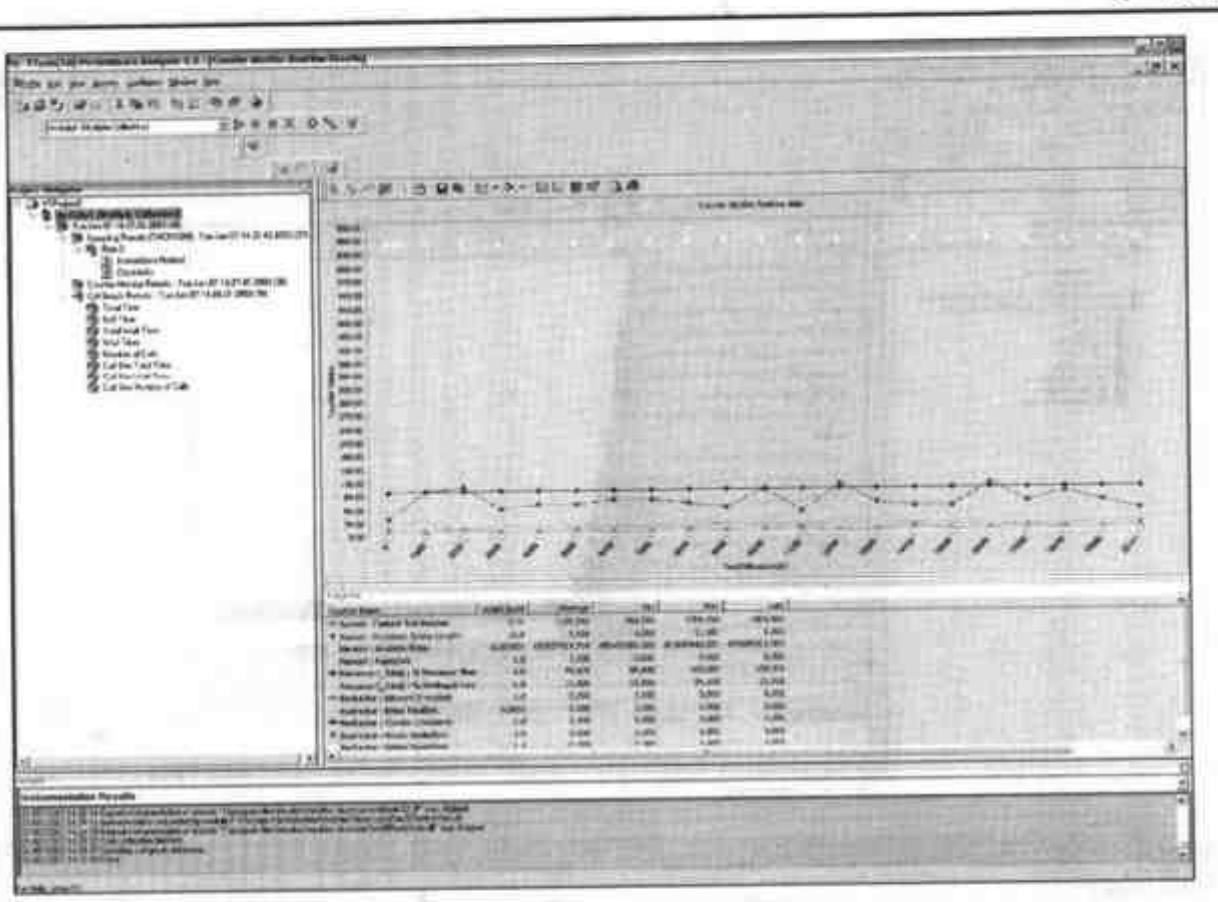


图 16.12　VTune 的主视图

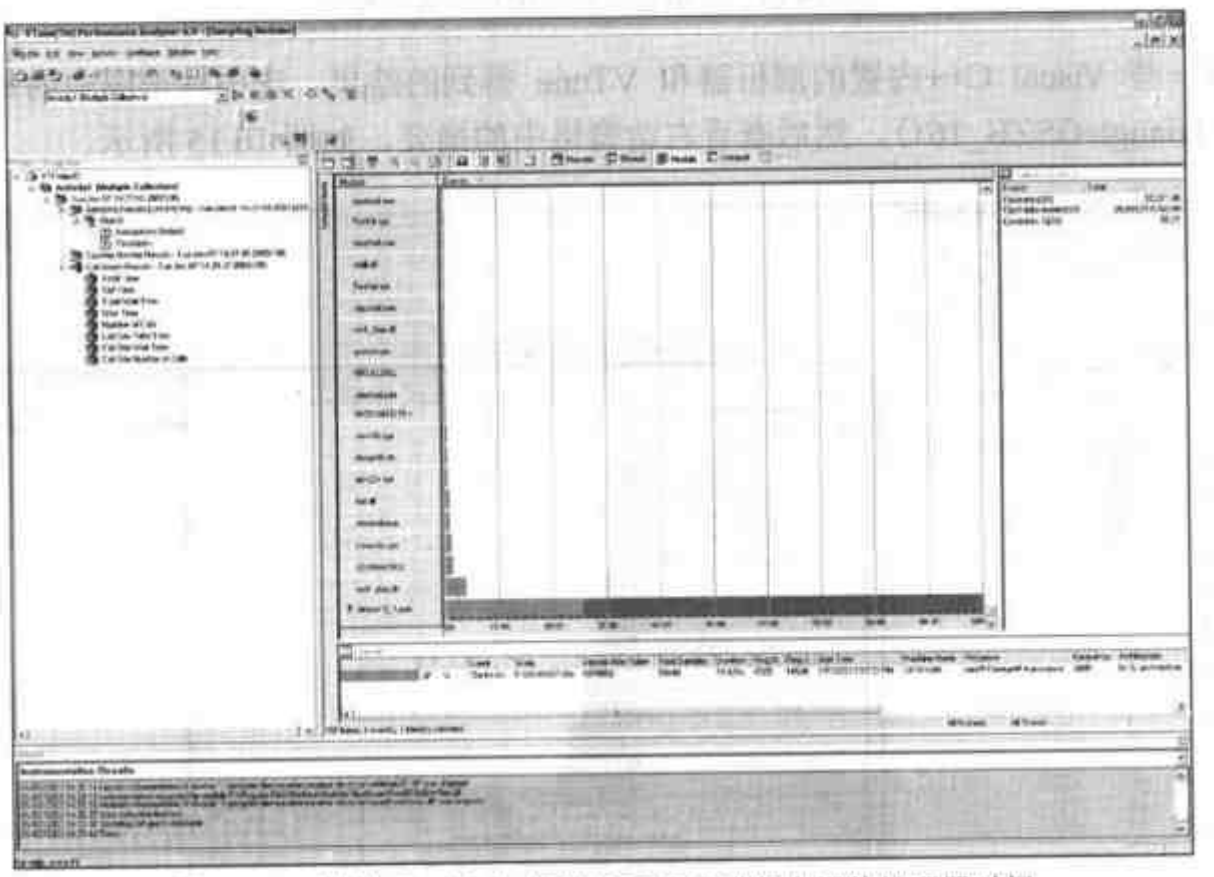


图 16.13　通过 Clockticks 视图查看每个进程占用的处理器时间

从这个视图中，可以看到每个函数占用的处理器时间。单击柱形统计图中的函数，它将呈高亮显示，同时最右边的 Selection Summary 窗格中的内容将相应地变化。该窗格列出了函数占用的时钟周期以及占用的处理器时间比例。双击函数将打开一个包含其源代码的窗口。关闭该窗口或按 Ctrl + Tab 键可回到函数列表视图。

现在，展示一些让您头晕目眩的东西。请看柱形图中从下往上数的第 5 个函数，在图 16.14 中，它呈高亮显示。这是将浮点数转换为整数的函数_ftol()。令人沮丧的是，这个运行阶段 C 函数占用的时间几乎与 3D 引擎代码一样长。这是被人们误解最深的数学问题之一。简单地说，像下面这样将一个 float 值赋给 int 变量：

```
float f;
int i = (int)f;
```

有时候编译器将生成一个函数调用，将浮点数转换为整数。因此，对于经过仔细优化的代码，这种开销所占的比例非常大。在很多图书中，作者简单地提到了这种问题，本章后面将介绍一些对此进行优化的方法。然而，重要的是，通过这个视图我们知道，如果能避免这个函数调用，可节省大约 5%的计算时间。

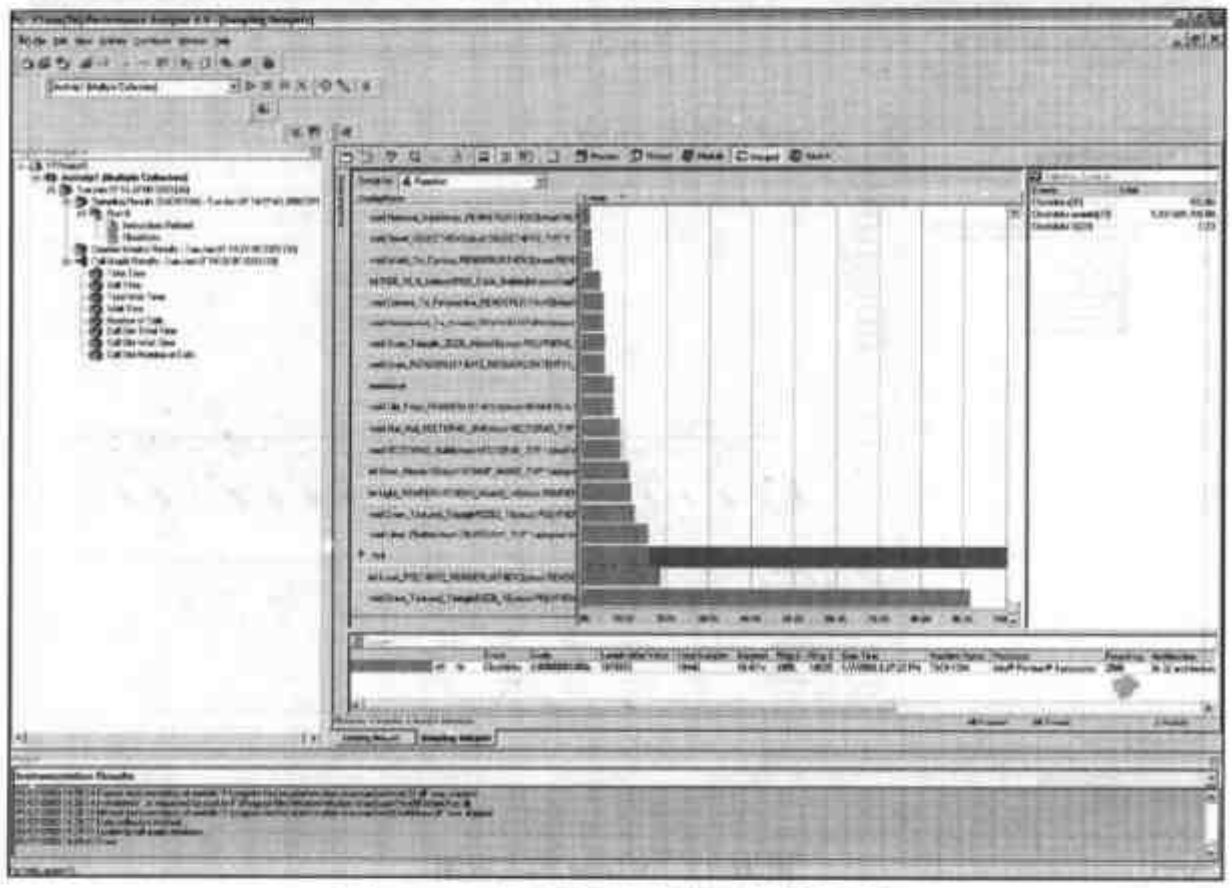


图 16.14　在函数级对应用程序进行分析

下面来比较一些 Visual C++内置的剖析器和 VTune 得到的结果。选中柱形图中的最后一个函数——Draw_Textured_TriangleGSZB_16()，然后查看右边窗格中的摘要，如图 16.15 所示。

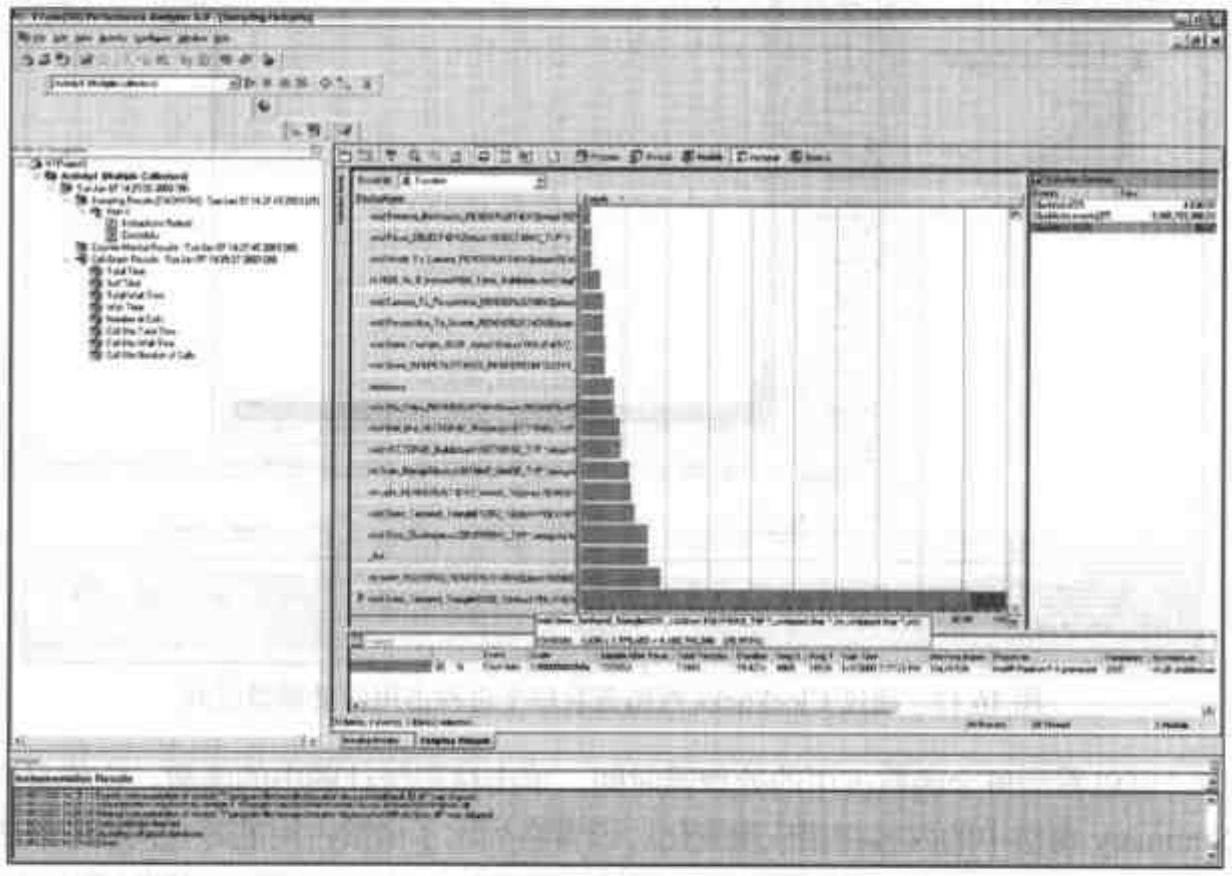


图 16.15　选择函数以查看其性能

最后一行是函数占用的时间比例，这里为 35.9%。如果回过头来看看 Visual C++提供的剖析信息，将发现该函数占用的时间比例为 21.3%。这两个数据之间存在一定的差别，其原因很多，如测试方法等，因此不用为此担心。使用该柱形图，可以详细了解每个函数的信息，一眼确定需要将优化重点放在哪里。当然，该视图还有很多其他的用途，但就现在而言，这足够了。要了解更详细的情况，VTune 教程是最好的参考资料。

（2）查看调用图

来看另一个非常重要的视图，它是基于程序的调用图（call graph），后者分析了函数调用情况。要查看调用图，可双击最左边的工程窗格中的 Call Graph Results，您将看到如图 16.16 所示的视图。这是一个图解树，让您能够查看每个函数调用流程——调用方和被调用方。单击函数后面的<和>可分别展开和折叠调

用列表。函数的颜色越红，它占用的时间越多。在图 16.16 中，应用程序为 thread_13A4。

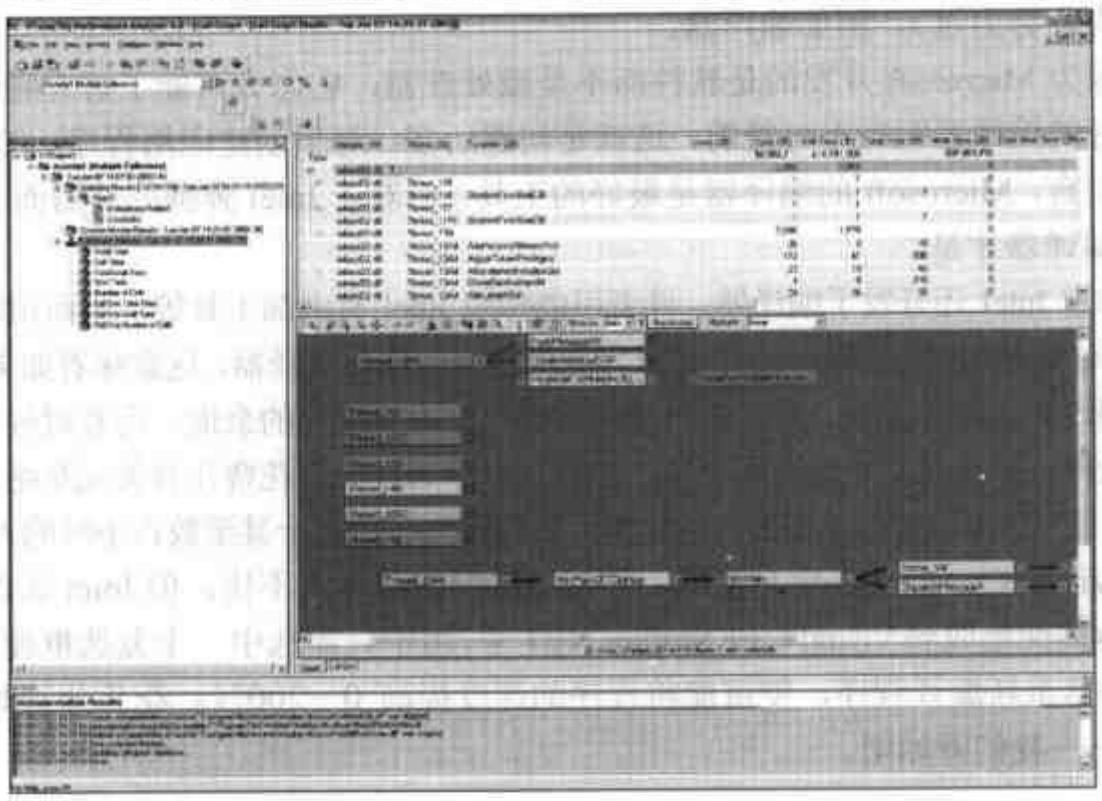


图 16.16　查看调用图

从图 16.17 可知，函数 Light_RENDERLIST4D()大量地调用了讨厌的_ftol()函数。事实上，很多时间都花在这个函数上。下面反过来，查看哪些函数调用了_ftol()。为此，可单击函数_ftol()左边的图标，结果如图 16.17 所示。这将显示_ftol()的进入（incoming）调用图，这样我们便知道需要在哪些函数中，使用其他编码技术来避免这种调用。

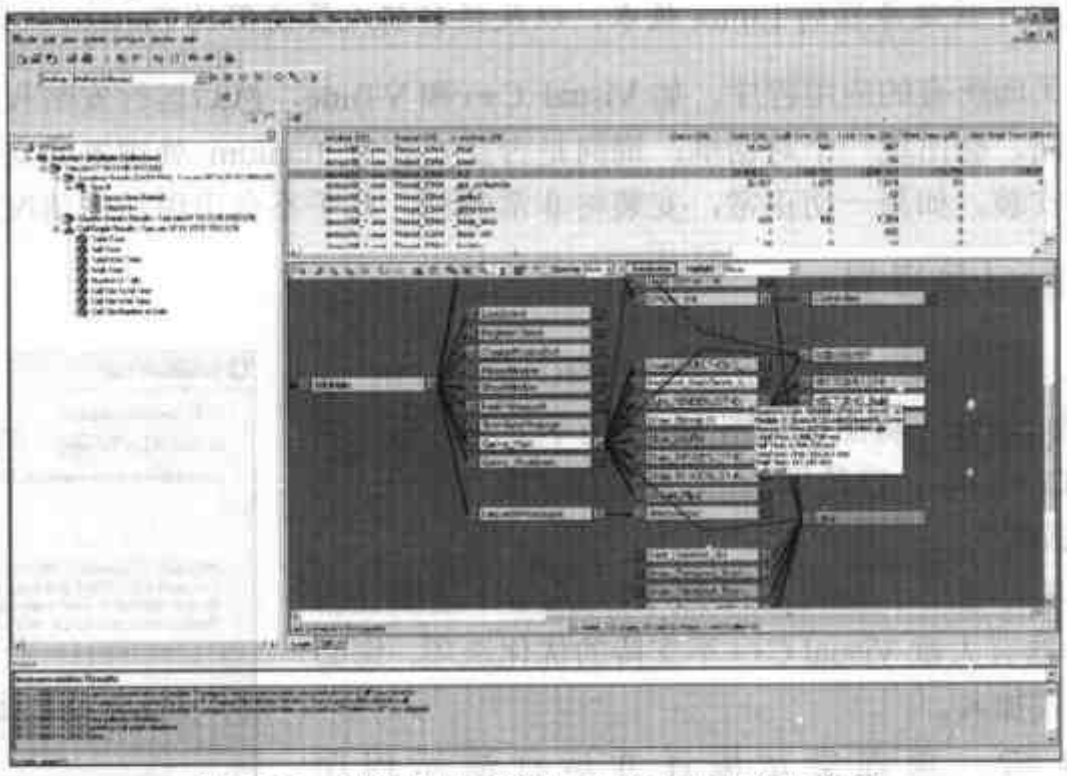


图 16.17　剖析光照函数执行时间如此之长的原因

总之，VTune 的强大功能令人难以置信。建议读者花几个小时来鼓捣它：尝试单击各种元素，看看会发生什么情况，并阅读 VTune 教程。

16.3　使用 Intel C++编译器

接下来要介绍的一项重要内容是 Intel 优化编译器。这里提到它的原因在于，95%的 PC 游戏都是使用 Visual C++编写的。这不难理解，因为 Microsoft 开发了优秀的工具，并将它们集成到了很多系统中。

尽管如此，有时候，其他公司为 PC 开发的编译器也被广泛使用。还记得 Borland 和 Watcom 吗？它们

由于无力与 Microsoft 竞争而倒闭或被兼并。然而，像 Microsoft 这样庞大的公司不可能在每个方面都处于领先水平（虽然它为此努力过），例如编译器。

这是合理的，因为 Microsoft 开发的是软件而不是微处理器，它怎么可能了解奔腾处理器的方方面面，进而开发编写出最好的编译器呢？不可能的，这就是问题所在。对于创建应用程序、.DLL 和 ActiveX 组件等典型的大型软件而言，Microsoft 的编译器是最好的工具，但对于 Intel 奔腾处理器而言，它并非很好的优化器，Intel 的优化编译器才是。

很多人甚至不知道 Intel 还开发了编译器。读者可能猜想 Intel 有内部工具等，但商用编译器呢？Intel 确实开发了编译器，而且对 Intel 处理器系列而言，是世界上最先进的优化编译器，这意味着如果在经济上承担得起。您可能想使用这种编译器来编译游戏。有时候，您的代码可能没有优化的余地，而有时候可能还有，但该编译器至少提供了一种选择；如果开发的是商用产品，为提高产品的性能而花费几百美元是绝对值得的。而要将性能提高相同的程度，使用 Visual C++手工进行优化时，可能需要花费数十甚至数百小时的人力。

经过多年使用 Visual C++后，再学习使用一种新工具的想法令人不快，但 Intel 认识到了这一点，使其 C++编译器能够作为插件集成到 Visual C++ Studio/.NET 中，用于只需选中一个复选框便可以使用该编译器。只需使用 Intel 编译器重新编译程序，便可能将程序的速度提高 0～200%。花几分钟时间来学习使用该编译器是绝对值得的——我们开始吧。

16.3.1 下载 Intel 的优化编译器

第一步是下载编译器本身。读者可以在 Intel 的网站中找到最新的版本，网址如下：

http://developer.intel.com/software/products/compilers/

在该网页中找到 Intel compiler for Windows free evaluation 链接，将该产品下载到硬盘中。

提示：Intel 还提供了该编译器的 Linux 版本，以及供科研人员使用的 Fortran 编译器。

下载编译器后，关闭所有的应用程序，如 Visual C++和 VTune，然后运行安装程序。安装过程与其他程序相同。在安装期间，会出现一个对话框，询问是否要安装对 Itanium 处理器的支持功能——请不要安装。另外请选择典型安装。如果一切正常，安装将非常顺利，几乎不会出现任何情况。

16.3.2 使用 Intel 编译器

启动 Visual C++，加载前面一直在使用的 DEMO16_1.CPP。选择菜单 Tools/Select Compiler，将出现如图 16.18 所示的对话框。选中复选框 Intel® C++ Compiler，以使用 Intel 优化编译器。然后单击 OK 按钮，关闭该对话框。

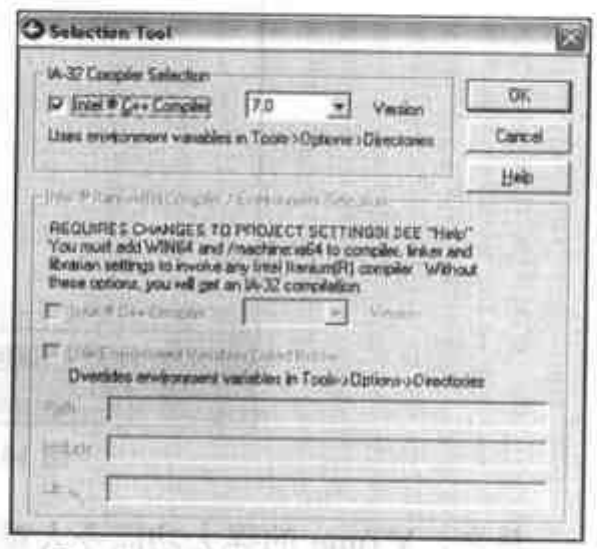


图 16.18 选择 Intel 编译器非常简单

99%的编译器选项都适用于 Intel 编译器，因此几乎不用做任何修改。然而 Intel 编译器有大量 Visual C++不支持的优化选项，您必须使用标记以手工方式加入。

这些将稍后介绍，现在尝试编译并运行演示程序 DEMOII16_1.CPP，看它是否能运行。选择菜单 Build/Rebuild All 以编译该工程。构建开始时，读者可能看到一些奇怪的警告或以前没有见过的编译器输出——这是件好事，它表明 Intel 编译器在发挥作用。

警告：编译时间可能是平常的 2～3 倍，因为 Intel 编译器将尝试使用各种优化策略。

如果编译成功，由于作者将“警告当作耳边风”的编程风格，将出现一些警告，但不会有错误。现在来运行该程序，看它能否正常运行。为此，选择菜单 Build/Execute...或按 Ctrl + F5 键，读者将看到工兵正在做自己的工作。如果读者足够敏感，将发现动画速度更快了——这是因为确实如此。根据您使用的硬件，演示程序的速度将提高 5～10%，这仅仅是通过重新编译实现的，太令人惊讶了！

16.3.3　使用编译器选项

Intel 优化编译器新增了大量优化特性，这些特性可以通过手工输入编译器标记来控制。表 16.1 列出了一些比较重要的标记。

表 16.1　　重要的 Intel 优化编译器选项

选项	含义
-G5	奔腾处理器
-G6	奔腾 Pro、奔腾 II 和奔腾 III 处理器
-G7	奔腾 4 和 Xeon 处理器
-Qxi	Intel 奔腾 Pro 和奔腾 II 处理器（使用 CMOV、FCMOV 和 FCOMI 指令）
-QxM	使用 MMX 指令（不包含 i 指令）的奔腾处理器
-QxK	使用流式 SIMD 扩展（意味着 i 指令和 M 指令）的奔腾 III 处理器
-QxW	使用流式 SIMD 扩展 2（意味着 i 指令、M 指令和 K 指令）的奔腾 4 和 Xeon 处理器
-Qaxi	Intel 奔腾 Pro 和奔腾 II 处理器（使用 CMOV、FCMOV 和 FCOMI 指令）
-QaxM	使用 MMX 指令的奔腾处理器
-QaxK	使用流式 SIMD 扩展（意味着 i 指令和 M 指令）的奔腾 III 处理器
-QaxW	使用流式 SIMD 扩展 2（意味着 i 指令、M 指令和 K 指令）的奔腾 4 和 Xeon 处理器
-Qax{i|M|K|W}	启用向量化器（vectorizer），生成专用和通用的 IA-32 代码（速度较慢）
-Qx{i|M|K|W}	启用向量化器（vectorizer），生成处理器专用的代码（速度较快）
-01	优化速度，但禁用一些会增加代码长度但速度提高不多的优化
-02	优化速度
-03	启用高级优化（不保证性能更高）
-0g	启用全局优化
-0s	启用大部分速度优化，但禁用会增加代码长度但速度提高不多的优化
-0t	启用所有速度优化

还有很多其他的编译器选项，但这些对初学者而言足够了。请针对自己使用的处理器设置编译器，逐渐提高优化等级，并尝试启用向量化选项。启用向量化选项后，可能使代码无法通过编译。在这种情况下，可以使用不同的选项来编译每个源文件，或者使用 Microsoft 编译器来编译它们。下面介绍如何对源文件指定要使用的编译器——Microsoft 编译器或 Intel 编译器。

16.3.4　手工为源文件选择编译器

假设在对话框 Select Compiler 中选择了 Intel 编译器，但某个源文件的错误太多，无法通过编译。要指定对某个源文件使用标准的 Microsoft 编译器，可采取如下步骤：

1. 在源文件树（source tree）窗口中，在该源文件上单击鼠标右键，然后选择 Settings。

2. 将出现如图 16.19 所示的对话框，单击选项卡 C/C++，然后在下拉式列表框 Category 中选择 General。

3. 在文本框 Preprocessor Definitions 中添加如下内容：

_USE_NON_INTEL_COMPILER

这样，Visual C++将使用 Microsoft 编译器而不是 Intel 编译器来编译该源文件。

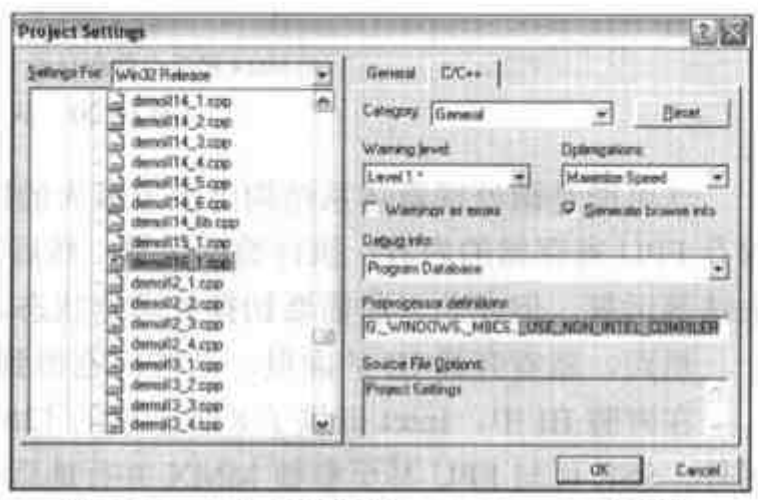


图 16.19　为源文件选择 Intel/Microsoft 编译器

提示：也可以设置为正常情况下使用 Microsoft 编译器，然后使用_USE_INTEL_COMPILER 来指定对某些源文件使用 Intel 编译器。

16.3.5 优化策略

安装好编译器并可以使用它来编译程序后，便可以做一切尝试：逐渐提高或降低优化等级，针对处理器设置优化选项，然后使用 Visual C++的剖析器或 VTune 来找出性能瓶颈。当然，这必须以一致、合理的方式进行——每次启用/关闭一个编译器选项，然后记录剖析结果。如果每次启用/关闭多个选项，将无法知道每个选项的效果，因此应每次修改一个选项，然后测试修改的效果。

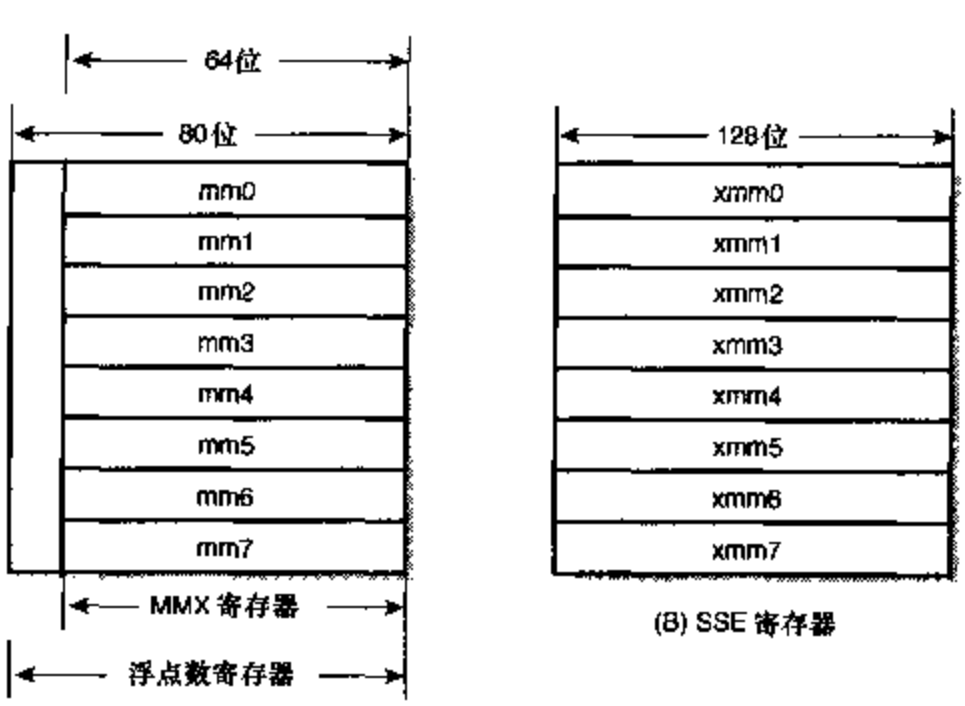
16.4 SIMD 编程初步

接下来要介绍的主题让作者相信基督再临，这就是 SIMD（单指令多数据）编程。如果读者听说并使用过 SIMD，可跳过本节；否则本节将提供震撼性启迪。

先来介绍一下术语和历史。Intel 在发布奔腾处理器的同时，发布了多媒体扩展（MMX）。Intel 对新的 MMX 技术寄托了改变世界、提高 3D 图形和 Internet 速度的厚望——正如读者知道的，这只不过是管用的市场宣传手法而已；如果再加上音乐和舞女，便是一个完整的市场宣传场景。

不幸的是，MMX 没有得到普及，辜负了 Intel 的厚望。其原因在于，MMX 是被硬塞到奔腾处理器的体系结构中的。基本上，MMX 技术只是在奔腾处理器的指令集中加入了一些指令，以支持平行（SIMD）整数运算。

问题是没有专供 MMX 使用的寄存器，因此 Intel 决定让 MMX 和浮点数公用寄存器，从分配给每个 FPU 寄存器的 80 位中拿出 64 位供 MMX 使用，如图 16.20a 所示。



(A) MMX 和浮点数寄存器　　(B) SSE 寄存器

图 16.20　从 FPU 栈中分配 MMX 寄存器

这可能是微处理器体系结构历史上最大的错误。问题在于，要使用 MMX 指令，必须切换处理器状态，保存 FPU 寄存器的内容，执行数学运算，然后恢复到原来的状态。因此，虽然使用 MMX 可以并行地执行加法等运算，但在此之前需要切换处理器状态，之后需要恢复到原样，因此 MMX 的效果并不好。

然而，随着奔腾 III 的面世，这些问题得到了妥善处理，并实现了流式 SIMD 扩展（SSE）。

在奔腾 III 中，Intel 提供了 8 个全新的 128 位寄存器，它们名为 XMM0～XMM7，如图 16.20b 所示。另外，SSE 可与 FPU 甚至整数 MMX 并行地运行，而不需要切换状态，保存寄存器内容。这些寄存器中的每个最多可存储 4 个 32 位的单精度浮点数，因此 SSE 是浮点数 SIMD，让您能够同时处理 4 个 32 位的浮点数。可以对这些浮点数执行加法、乘法、求平方根、比较等运算。事实上，在指令集中新增了大约 70

个 SSE 指令，这相当于新增了一个处理器。

深入介绍 SSE 之前，这里有必要提一下流式 SIMD 扩展第 2 版（SSE2）。这是奔腾 4 处理器新增的特性，它新增了 150 个指令，这些指令支持双精度而不是单精度浮点数运算。如果读者使用的是奔腾 4 处理器，将可以并行地对两个 64 位的双精度浮点数（而不是 4 个 32 位的单精度浮点数）执行两种运算。图 16.21 说明了这一点。

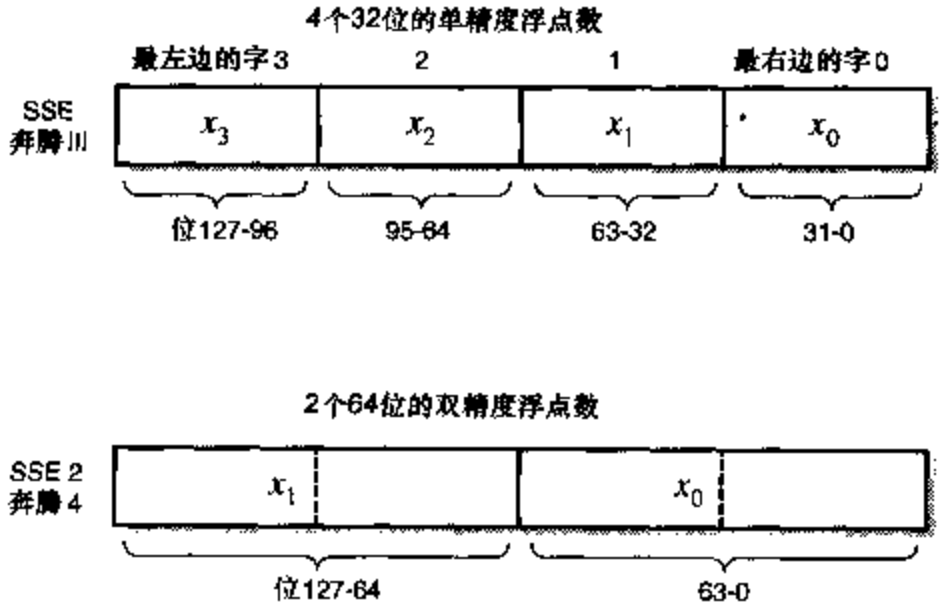


图 16.21　128 位 SSE/SSE2 寄存器的分解

SSE2 很好，但除非进行的是需要 64 位精度的科学编程或类似的工作，否则没有必要使用它。在作者看来，在游戏中还没有必要使用双精度值，作者宁愿使用 SSE 同时执行四次运算，而不想为将精度提高一倍而使用 SSE2 同时执行两次运算。奔腾 4 支持 SSE 和 SSE2，因此为奔腾 III 编写的代码也可以在奔腾 4 上正确地运行。

另外，在 SSE 和 SSE2 实现中，SIMD 寄存器 XMM0～XMM7 都是 128 位，因此工作方式完全相同。这里只讨论奔腾 III 中的 SSE，换句话说，我们只想并行地对 32 位的单精度值进行 4 路 SIMD 处理。

从现在开始，作者将 SSE/SSE2 简称为 SIMD。要使用下面将介绍的技术，读者使用的处理器必须是奔腾 III 或更好的。

16.4.1　SIMD 基本体系结构

SIMD 基本体系结构由 8 个 128 位的寄存器组成，如图 16.20 所示。每个寄存器可存储 4 个 32 位的单精度值，并行地对这些值执行运算。每个 SIMD 寄存器（XMM0～XMM7）的使用情况如图 16.21 所示：第 0～31 位用于存储最左边的元素，第 96～127 位用于存储最右边的元素。每个元素对其他元素一无所知，并行地对这些元素执行运算，因此有术语单指令、多数据。

SIMD 尤其适用于 3D 图形学，因为很多运算都能够并行进行，如点积、矩阵乘法、光照处理、光栅化等。然而，也不要抱过高的期望，因为虽然使用 SIMD 可以同时执行 4 次浮点数运算，但将数据存入寄存器和从寄存器中取出以及在程序中将数据转换为适合于 SIMD 的格式都会降低性能。在最理想的情况下，性能可提高大约 200%，通过合理的 SIMD 编程，性能通常能提高 150～170%。换句话说，在大多数情况下，数学运算代码的速度至少可提高一倍。

16.4.2　使用 SIMD

知道步骤后，使用 SIMD 将非常容易，下面来介绍使用步骤。首先，除了处理器必须是奔腾 III+外，还必须使用支持 SIMD 的操作系统。在大多数情况下，Windows 2000、Windows XP 及更高的版本都支持 SSE/SSE2。要检查 Windows 操作系统是否支持 MMX 和 SSE，可使用下述代码：

```
if (IsProcessorFeaturePresent(PF_MMX_INSTRUCTIONS_AVAILABLE))
  printf("\nMMX Available.\n");
else
  printf("\nMMX NOT Available.\n");
```

```
if (IsProcessorFeaturePresent(PF_XMMI_INSTRUCTIONS_AVAILABLE))
  printf("\nXMMI Available.\n");
else
  printf("\nXMMI NOT Available.\n");
```

上述代码使用 Windows API IsProcessorFeaturePresent() 来查询指定的特性，该函数返回 TRUE 或 FALSE。通过该函数还可以查询很多其他的特性，建议读者参阅 API 帮助信息。

知道 OS 和处理器支持 SSE（PF_XMMI_INSTRUCTIONS_AVAILABLE）后，需要编写一些代码来使用它。当然，可以对 Intel 编译器进行设置，使之针对奔腾 III 和 4 处理器对代码进行优化，这样优化期间它将使用 SIMD 代码。然而，我们感兴趣的是自己编写使用 SIMD 的代码，下面来看看为此需要如何做。

1. 设置编译器使之支持 SIMD

Intel 优化处理器支持 SIMD，且包含合适的头文件和库模块。然而，如果使用 Microsoft Visual C++中的编译器，必须下载新的处理器 pack upgrade。这将安装所有的头文件、库模块等。您的 Visual C++必须是专业版或企业版，并安装 Service Pack upgrade 4 或 5。处理器 Service Pack 可从下述网址下载：

http://msdn.microsoft.com/vstudio/downloads/tools/ppack/default.asp

提示：安装处理器 Service Pack 时，将同时安装 MASM。

强烈推荐读者安装最新的 Visual C++ Studio Service Pack，其下载网址如下：

http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/default.asp

提示：如果读者没有 Visual C++专业版或企业版，但又想尝试使用 SIMD 指令，也不用担心：Intel 编译器内置了对 SIMD 指令的支持，因此只需下载该编译器的免费评估版（至少可使用 30 天）即可。

下载处理器 Service Pack 后，按说明安装它即可。安装完成后，系统可能询问是否重新启动计算机。至此，准备工作便已经就绪。

下面使用一个简单的程序来测试新的处理器 Service Pack（SIMD 支持）。首先创建一个简单的控制台应用程序，然后输入如下所示的 DEMOII16_2.CPP 中的代码（也可以使用附带光盘中的源文件）：

```
// DEMOII16_2.CPP - Hello world SIMD demo

// I N C L U D E S ///////////////////////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN
#include <windows.h>   // 包含重要的 windows 功能
#include <windowsx.h>
#include <stdio.h>
#include <math.h>
#include <xmmintrin.h> // 以支持 SIMD(SSEI)

// MAIN //////////////////////////////////////////////////////////////////////

void main()
{
// 打印处理器和操作系统对 SIMD 的支持情况
if (IsProcessorFeaturePresent(PF_MMX_INSTRUCTIONS_AVAILABLE))
  printf("\nMMX Available.\n");
else
  printf("\nMMX NOT Available.\n");

if (IsProcessorFeaturePresent(PF_XMMI_INSTRUCTIONS_AVAILABLE))
  printf("\nXMMI Available.\n");
else
  {
  printf("\nXMMI NOT Available.\n");
  return;
  } // end if

// 定义 3 个 SIMD 封装值，必须是与 16 字节边界对齐的
__declspec(align(16)) static float x[4] = {1,2,3,4};
```

实际上，SIMD 使用封装数据类型：4 个 32 位的浮点数被紧密的封装在一起，起始位置为 16 字节边界。您可以使用内置的数据类型__m128，也可以使用 16 字节对齐的 float[4]数组，后者可能更方便，因此这样可以使用索引访问其中每个元素。然而，如果使用内置的本征库来实现 SIMD，必须使用数据类型__m128 或进行数据类型转换。

使用 SIMD 时需要做出的下一个重要决策是，使用哪种数据结构。SIMD 能够同时处理 4 个数据元素，对其执行简单的逻辑或数学运算。然而，由于 SIMD 的效率非常高，您可能发现，将数据转换为 SIMD 封装字以及从计算结果中提取数据会浪费时间。因此，定义数据结构时，应考虑使用结构数组（array-of-structures，AoS）还是数组结构（structure-of-arrays，SoA），如图 16.23 所示。

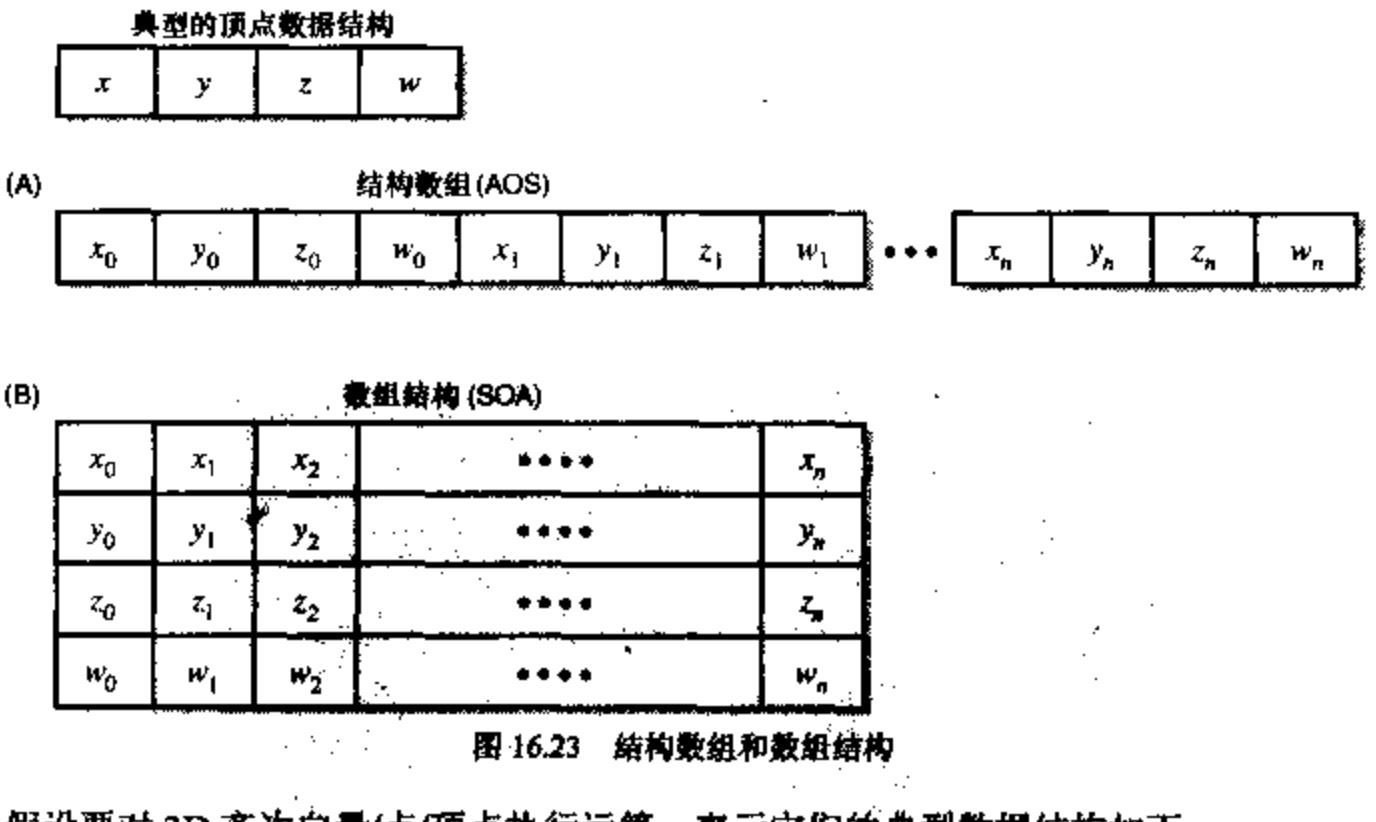


图 16.23 结构数组和数组结构

例如，假设要对 3D 齐次向量/点/顶点执行运算，表示它们的典型数据结构如下：

```
typedef struct VECTOR4D_TYP
{
__declspec(align(16)) float x,y,z,w;
} VECTOR4D, *VECTOR4D_PTR;
```

使用这种结构，可以像下面这样定义 1000 个顶点：

```
VECTOR4D vertex_list[1000];
```

这是一个结构数组。现在，假设要将所有的顶点相加，这很简单：将一个 VECTOR4D 加载到一个 SIMD 寄存器中，然后将下一个 VCTOR4D 与之相加，并存储结果；重复上述过程，直到将 1000 个顶点都相加为止。

现在暂时不考虑细节，看看我们所做的工作。我们将一个 VECTOR4D 以 $x_iy_iz_iw_i$（从右到左）格式加载到寄存器中，然后将其与 $x_jy_jz_jw_j$ 相加。然而，我们计算了元素 w，这是在浪费时间。w 用于其他目的，将向量相加时，在大多数情况下总是假定 w 为 1.0。这个例子表明，也许另一种格式更合适。

使用数组结构可解决这种问题。我们不是定义一个顶点数据结构，然后创建一个这种结构的数组，而是这样做：

```
typedef struct VERTEXLIST4D_TYP
{
__declspec(align(16)) float x[1000];
__declspec(align(16)) float y[1000];
__declspec(align(16)) float z[1000];
__declspec(align(16)) float w[1000];
} VERTEXLIST4D, * VERTEXLIST4D_PTR;
```

然后，在循环中，我们每次加载 4 个 x 值，将其同下一组 x 值相加，这将循环 1000/4 次。对于 y 和 z 做相同的处理。

实际上，SIMD 使用封装数据类型：4 个 32 位的浮点数被紧密的封装在一起，起始位置为 16 字节边界。您可以使用内置的数据类型__m128，也可以使用 16 字节对齐的 float[4]数组，后者可能更方便，因此这样可以使用索引访问其中每个元素。然而，如果使用内置的本征库来实现 SIMD，必须使用数据类型__m128 或进行数据类型转换。

使用 SIMD 时需要做出的下一个重要决策是，使用哪种数据结构。SIMD 能够同时处理 4 个数据元素，对其执行简单的逻辑或数学运算。然而，由于 SIMD 的效率非常高，您可能发现，将数据转换为 SIMD 封装字以及从计算结果中提取数据会浪费时间。因此，定义数据结构时，应考虑使用结构数组（array-of-structures，AoS）还是数组结构（structure-of-arrays，SoA），如图 16.23 所示。

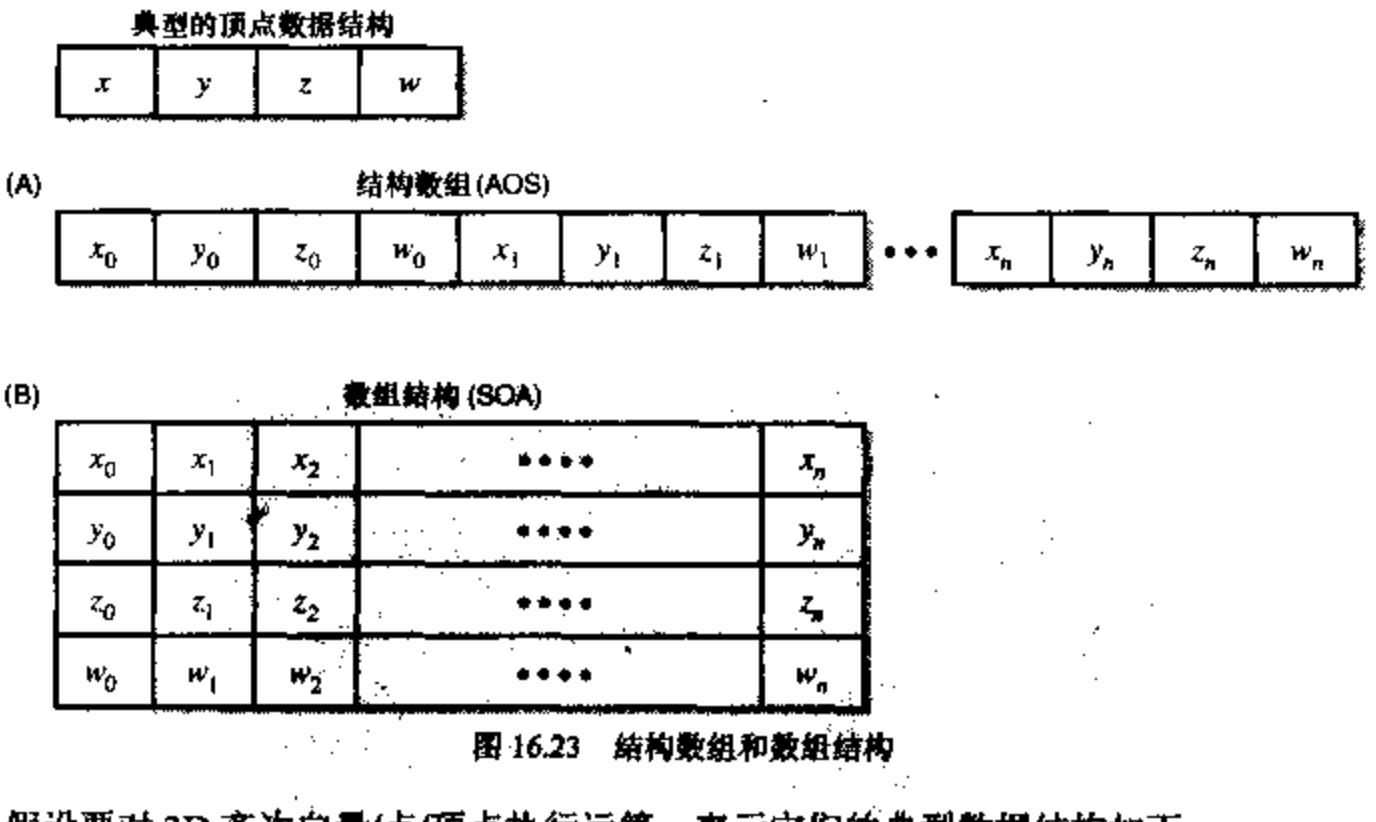


图 16.23 结构数组和数组结构

例如，假设要对 3D 齐次向量/点/顶点执行运算，表示它们的典型数据结构如下：

```
typedef struct VECTOR4D_TYP
{
__declspec(align(16)) float x,y,z,w;
} VECTOR4D, *VECTOR4D_PTR;
```

使用这种结构，可以像下面这样定义 1000 个顶点：

```
VECTOR4D vertex_list[1000];
```

这是一个结构数组。现在，假设要将所有的顶点相加，这很简单：将一个 VECTOR4D 加载到一个 SIMD 寄存器中，然后将下一个 VCTOR4D 与之相加，并存储结果；重复上述过程，直到将 1000 个顶点都相加为止。

现在暂时不考虑细节，看看我们所做的工作。我们将一个 VECTOR4D 以 $x_iy_iz_iw_i$（从右到左）格式加载到寄存器中，然后将其与 $x_jy_jz_jw_j$ 相加。然而，我们计算了元素 w，这是在浪费时间。w 用于其他目的，将向量相加时，在大多数情况下总是假定 w 为 1.0。这个例子表明，也许另一种格式更合适。

使用数组结构可解决这种问题。我们不是定义一个顶点数据结构，然后创建一个这种结构的数组，而是这样做：

```
typedef struct VERTEXLIST4D_TYP
{
__declspec(align(16)) float x[1000];
__declspec(align(16)) float y[1000];
__declspec(align(16)) float z[1000];
__declspec(align(16)) float w[1000];
} VERTEXLIST4D, * VERTEXLIST4D_PTR;
```

然后，在循环中，我们每次加载 4 个 x 值，将其同下一组 x 值相加，这将循环 1000/4 次。对于 y 和 z 做相同的处理。

这些问题与您要使用 SIMD 来做什么相关。例如，如果要计算大量的点积，则 SOA 格式更合适；但如果要大量的 32 位字相加，则 AOS 的效果可能更好。关键之处在于，这是在 SIMD 编程过程中设计数据结构时需要考虑的一个重要因素。例如，对于前面将向量相加的范例，SOA 数据结构的速度更快，但这对引擎存储数据来说并没有帮助。

3. 一些简单的 SIMP 范例

SIMD 是一个很大的主题。事实上，它基本上是一种全新的语言和处理器汇编语言，因为它支持的指令有 70 多个。在几页的篇幅中无法做全面介绍，这里只阐述如何使用 SIMD 执行基本的数学运算。另外，在本节的最后，将列出 Intel 的本征函数。

本征函数基本上是 SIMD 汇编语言指令封装函数（wrapper），使用本征函数可能比直接使用 SIMD 汇编语言简单，但速度稍微慢些，因此您可能坚持使用汇编语言。不用说，要执行下述范例中的代码，必须安装处理器 Service Pack（如果使用 Visual C++编译器）或者使用 Intel 编译器。另外，还必须包含下述头文件：

```
#include <xmmintrin.h>
```

提示：如果要支持奔腾 4 SSE2（双精度 SIMD），还必须包含头文件 emmintrin.h。

最后，这里只介绍了一部分指令，它们是每种主要指令中的代表。

对于所有的范例，假设有如下声明：

```
__declspec(align(16)) float x[4], y[4], z[4];
__m128 m0, m1, m2;
```

（1）数据移动和混合（shuffling）

使用汇编语言或本征函数将 SIMD 值移到 SIMD 寄存器中的方法有很多。一般而言，内嵌汇编器知道每个变量的类型以及指针和值之间的差别。因此，您总是可以直接使用变量来存储 SIMD 值，然后使用单精度移动-对齐指令 movaps 将其赋给 XMM 寄存器。该指令有两种形式：

```
movasp xmm_dest, xmm_src/memory
movasp xmm_dest/memory, xmm_src
```

基本上，总是可以在两个 XMM 寄存器之间移动 SMID 值；另外，还可以在内存和 XMM 寄存器之间移动 SMID 值；但不能在内存之间移动。下面来看一些范例。

范例：将 x[ ]移到 xmm0 中：

```
_asm
   {
   movaps xmm0, x
   }
```

还可以使用指针强制类型转换来避免源操作数导致编译器警告：

```
_asm
   {
   movaps xmm0, XMMWORD PTR x
   }
```

当然，也可以使用数据指针 esi、edi、edx 等，如下所示：

```
_asm
   {
   lea esi, x
   movaps xmm0, [esi]
   }
```

同样，要存储 XMM 寄存器中的结果，只需使用一个指向目标内存的指针即可。

范例：将 xmm0 移到 z[ ]中：

```
_asm
   {
   movaps z, xmm0
```

```
}
```

下面是用于存储和设置寄存器的本征函数：

```
// 将 SIMD 本征类型__m128 值存储到浮点数组*v 中
void _mm_store_ps(float *v, __m128 a);

// 将使用 a、b、c、d 给本征类型__m128 变量赋值
__m128 _mm_set_ps(float a, float b, float c, float d);
```

范例：将 m0 设置为[1，2，3，4]：

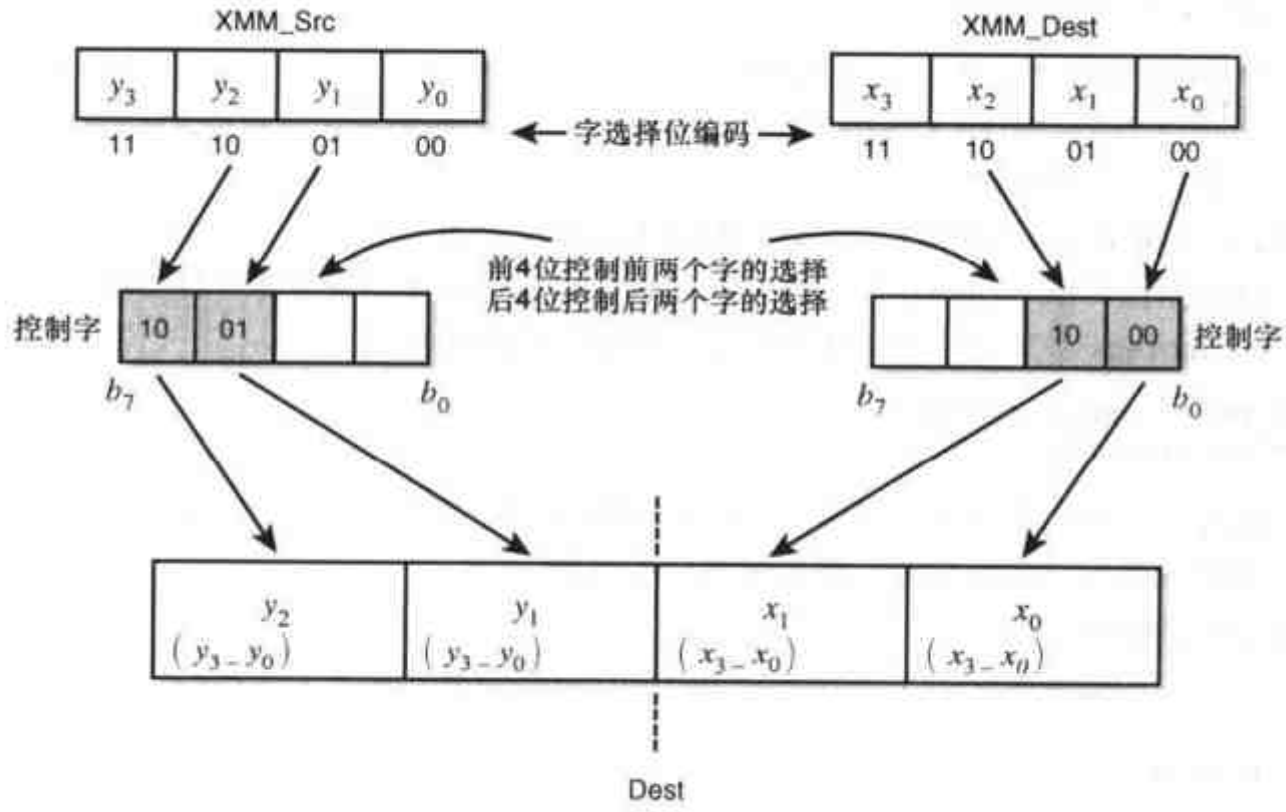
```
m0 = _mm_set_ps(1, 2, 3, 4);
```

范例：将 m0 中的值存储到 x[ ]中：

```
_mm_store_ps(x, m0);
```

当然，还有很多用于移动、初始化和赋值的本征函数，但上述函数对起步而言足够了。

现在，读者能够移动数据、将 SIMD 数据赋给寄存器。下面来看看使用 SIMD 时必须掌握的一项内容：数据混合——将字移到同一个 SIMD 寄存器的其他位置。在 SIMD 指令集中，用于执行这些操作的 SIMD 指令有很多，这里只介绍最有代表性的指令 shufps，图 16.24 对其做了说明。



ShuFps XMM_dest, XMM_Src, control8

图 16.24　数据移动指令 shufps

简单地说，该指令有三个操作数：源、目标和 8 位的控制字。该指令从目标操作数和源操作数中分别取出两个字，然后将它们存储到目标操作数中。因此，目标操作数也被用作源操作数。可将这种操作视为由两步组成：首先，将两个操作数都视为源操作数；然后最终结果覆盖其中的一个操作数——第一个操作数，即指令原型中的 xmm_dest。下面简要地讨论一下这个指令的细节。

xmm_dest 是目标 SIMD 寄存器，xmm_scr 是源 SIMD 寄存器。比较棘手的部分是该指令的工作原理（更详细的信息，请参阅 Intel 用户手册）。

基本上，该指令从源寄存器中取出两个字，将其作为目标寄存器的前两个字；并从目标寄存器中取出两个字节，将其作为目标寄存器的后两个字（参见图 16.24）。因此，正如前面指出的，目标寄存器既是源寄存器也是目标寄存器——但应该这样想，首先将两个寄存器是视为源寄存器，然后最终结果被存储到其中一个源寄存器中。表 16.2 说明了选择控制字的位编码，它们指定了从源寄存器或目标寄存器中取出哪个 32 位字。

表 16.2　　shufps 中的控制字编码

两位值	含义	两位值	含义
00	选择第 0 个字（最右边的字，第 0～31 位）	10	选择第 2 个字（第 64～95 位）
01	选择第 1 个字（第 32～63 位）	11	选择第 3 个字（最左边的字，第 96～127 位）

控制字 control8 的编码如下：

A_1A_0　B_1B_0　C_1C_0　D_1D_0

其中每个两位的子字指定了从源操作数中取出哪个字，表 16.3 描述了这些子控制字的编码。

表 16.3　　两位控制子字的编码

控制子字	用途
D_1D_0	指定将目标操作数中的哪个 32 位字作为最终结果的第 0 个字
C_1C_0	指定将目标操作数中的哪个 32 位字作为最终结果的第 1 个字
B_1B_0	指定将源操作数中的哪个 32 位字作为最终结果的第 2 个字
A_1A_0	指定将目标操作数中的哪个 32 位字作为最终结果的第 3 个字

使用指令 shufps 时，需要根据如何移动字来指定控制字，这很容易出错。为此，使用下述宏会有所帮助：

```
// srch - A1A0
// srcl - B1B0

// desth - C1C0
// destl - D1D0
#define SIMD_SHUFFLE(srch,srcl,desth,destl) (((srch) << 6) | ((srcl) << 4)
                    | ((desth) << 2) | ((destl)))
```

基本上，上述宏接受 4 个参数（源操作数和目标操作数的选择控制子字），通过移位将它们放到合适的位置；注释指出了各个参数对应的两位控制子字。

（2）通用的 SIMD 运算流程

接下来的几小节将讨论 SIMD 能够执行的一些基本运算。一般而言，它们遵循相同的并行流程，如图 16.25 所示。也就是说，一种运算被应用于两个 SIMD 源寄存器，其中一个源寄存器被用作最终的目标寄存器。

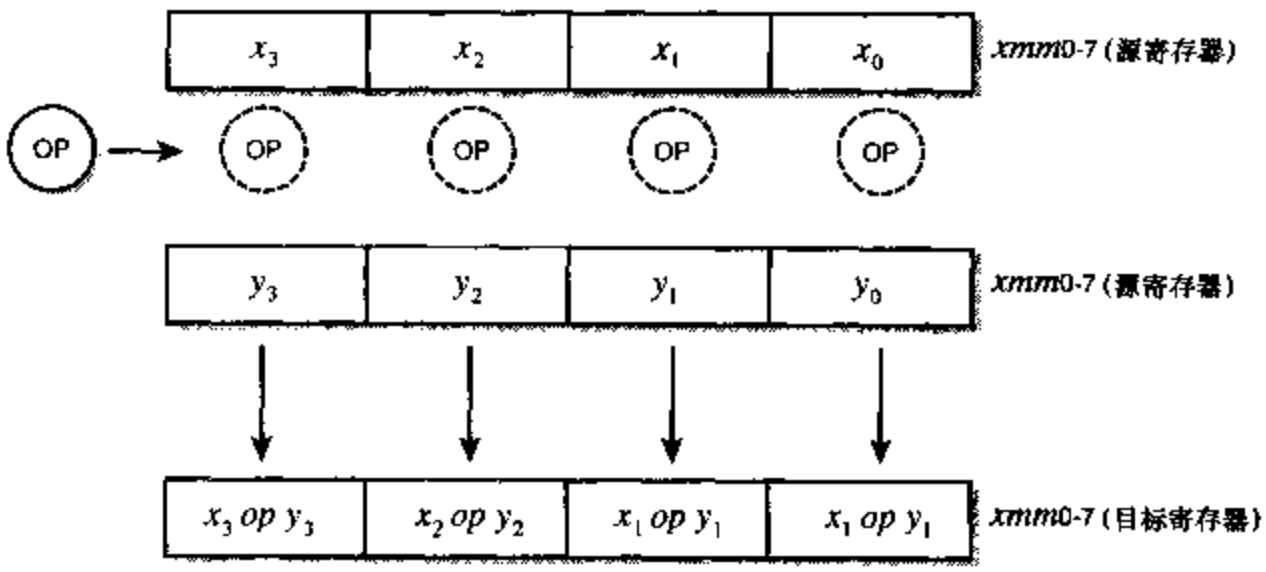


图 16.25　通用的 SIMD 运算流程

（3）加法和减法

SIMD 加法和减法指令并行地对 4 对 32 位值进行相加和相减。先来介绍加法指令：

```
addps xmm_dest, xmm_src/memory
```

addps 将一个 XMM 寄存器或位于内存中的 128 位操作数加入到目标 XMM 寄存器中。

范例：计算 x0+y0、x1+y1、x2+y2、x3+y3：

```
_asm {
   movaps xmm0, x // 将 x 的内容移到寄存器 xmm0 中
   addps xmm0, y // 将 y 的内容加入到寄存器 xmm0 中
   movaps z, xmm0 // 将结果(x0+y0, x1+y1, x2+y2, x3+y3)存储到 z 中
   }
```

加法指令的格式如下：

```
subps xmm_dest, xmm_src/memory
```

subps 从目标 MMX 寄存器中减去一个 XMM 寄存器或位于内存中的 128 位操作数。

范例：计算 x0-y0、x1-y1、x2-y2、x3-y3：

```
_asm {
   movaps xmm0, x // 将 x 的内容移到寄存器 xmm0 中
   subps xmm0, y // 将寄存器 xmm0 减去 y 的内容
   movaps z, xmm0 // 将结果(x0-y0, x1-y1, x2-y2, x3-y3)存储到 z 中
   }
```

（4）乘法和除法

SIMD 乘法和除法指令并行地对 4 对 32 位的值相乘和相除。先来介绍乘法指令：

```
mulps xmm_dest, xmm_src/memory
```

mulps 将一个 MMX 寄存器或位于内存中的 128 位操作数与目标 XMM 寄存器相乘。

范例：计算 x0*y0、x1*y1、x2*y2、x3*y3：

```
_asm {
   movaps xmm0, x // 将 x 的内容移到寄存器 xmm0 中
   mulps xmm0, y // 用 y 的内容乘以寄存器 xmm0
   movaps z, xmm0 // 将结果(x0*y0, x1*y1, x2*y2, x3*y3)存储到 z 中
   }
```

除法指令的格式如下：

```
divps xmm_dest, xmm_src/memory
```

divps 用一个 MMX 寄存器或位于内存中的 128 位操作数去除目标 XMM 寄存器。

范例：计算 x0/y0、x1/y1、x2/y2、x3/y3：

```
_asm {
   movaps xmm0, x // 将 x 的内容移到寄存器 xmm0 中
   divps xmm0, y // 用 y 的内容去除 xmm0 寄存器
   movaps z, xmm0 // 将结果(x0/y0, x1/y1, x2/y2, x3/y3)存储到 z 中
   }
```

（5）平方根

SIMD 平方根指令并行地计算 4 个 32 位值的平方根：

```
sqrtps xmm_dest, xmm_src/memory
```

sqrtps 计算位于 XMM 寄存器中或 128 位操作数中的 4 个 32 位值的平方根，然后将结果存储到目标 XMM 寄存器中。

范例：计算 $x0^{1/2}$、$x1^{1/2}$、$x2^{1/2}$、$x3^{1/2}$

```
_asm {
   sqrtps xmm0, x // 计算 x0、x1、x2、x3 的平方根
   movaps z, xmm0 // 将结果(x0^1/2, x1^1/2, x2^1/2, x3^1/2)存储到 z 中
   }
```

（6）逻辑运算

SIMD 支持各种逻辑运算。基本的逻辑指令 AND、OR 和 XOR 处理的都是封装的、单精度 32 位值，

它们的格式如下：

```
andps xmm_dest, xmm_src/memory
orps xmm_dest, xmm_src/memory
xorps xmm_dest, xmm_src/memory
```

每种逻辑运算对内存操作数或 XMM 寄存器和目标 XMM 寄存器执行相应的运算。结果存储在目标寄存器中。

范例：计算 z = ((x AND y) OR z)

```
_asm {
   mov xmm0, x // 将 x 的内容移到寄存器 xmm0 中
   mov xmm1, y // 将 y 的内容移到寄存器 xmm1 中
   mov xmm2, z // 将 z 的内容移到寄存器 xmm2 中

   andps xmm0, xmm1 // xmm0 = x AND y
   orps xmm0, xmm2 // xmm0 = (x AND y) OR z
   movps z, xmm0  // z = xmm0
   }
```

（7）比较运算

最后，还有很多比较指令，其中 cmpps 指令与 IA-32 标准指令 cmp 相同，其格式如下：

```
cmpps xmm_dest, xmm_src, immediate_8
```

分别对 4 对 32 位封装值进行比较，结果为 4 个 32 位的值（根据比较结果，每个值中的 32 位全为 0 或 1），存储在 xmm_dest 中。immediate_8 指定如何进行比较，可能的比较方式如表 16.4 所示。

表 16.4　　比较运算符

第 0～2 位的值	比较方式	第 0～2 位的值	比较方式
0	等于	4	不等于
1	小于	5	不小于
2	小于等于	6	大于
3	非排序比较（Unordered）	7	排序比较（ordered）

注：第 4～7 位被保留。

范例：检测 x[ ]是否小于 y[ ]：

```
-asm {
   mov xmm0, y
   cmpps xmm0, x
   //根据 xmm0 中的比较结果，执行相应的代码
   }
```

（8）本征函数

本征函数与指令极其相似。要了解本征函数，最好的方法是详细研读文件 xmmintrin.h。下面列出了一些重要的本征函数。

算术运算本征函数：

```
__m128 _mm_add_ps(__m128 a, __m128 b); // a+b
__m128 _mm_sub_ps(__m128 a, __m128 b); // a-b
__m128 _mm_mul_ps(__m128 a, __m128 b); // a*b
__m128 _mm_div_ps(__m128 a, __m128 b); // a/b
__m128 _mm_sqrt_ps(__m128 a); // a 的平方根
__m128 _mm_rcp_ps(__m128 a); // a 的倒数
__m128 _mm_rsqrt_ps(__m128 a); // a 的平方根的倒数
__m128 _mm_min_ps(__m128 a, __m128 b); // min(a,b)
__m128 _mm_max_ps(__m128 a, __m128 b); // max(a,b)
```

范例：m0 = m1 + m2

```
m0 = _mm_add_ps(m1, m2);
```

逻辑运算本征函数：

```
__m128 _mm_and_ps(__m128 a, __m128 b); // a AND b
__m128 _mm_andnot_ps(__m128 a, __m128 b); // a NAND b
__m128 _mm_or_ps(__m128 a, __m128 b); // a OR b
__m128 _mm_xor_ps(__m128 a, __m128 b); // a XOR b
```

范例：m0 = m1 AND m2：

```
m0 = _mm_and_ps(m1, m2);
```

比较运算本征函数：

```
__m128 _mm_cmpeq_ps(__m128 a, __m128 b); // a == b
__m128 _mm_cmplt_ps(__m128 a, __m128 b); // a < b
__m128 _mm_cmple_ps(__m128 a, __m128 b); // a <= b
__m128 _mm_cmpgt_ps(__m128 a, __m128 b); // a > b
__m128 _mm_cmpge_ps(__m128 a, __m128 b); // a >= b
__m128 _mm_cmpneq_ps(__m128 a, __m128 b); // a!=b
__m128 _mm_cmpnlt_ps(__m128 a, __m128 b); // !(a < b)
__m128 _mm_cmpnle_ps(__m128 a, __m128 b); // !(a <= b)
__m128 _mm_cmpngt_ps(__m128 a, __m128 b); // !(a > b)
__m128 _mm_cmpnge_ps(__m128 a, __m128 b); // !(a >= b)
__m128 _mm_cmpord_ps(__m128 a, __m128 b);
__m128 _mm_cmpunord_ps(__m128 a, __m128 b);
```

范例： m0 = m1 < m2：

```
m0 = _mm_ cmplt_ps(m1, m2);
```

使用本征函数还是内嵌汇编语言由您决定，然而必须牢记的是，很多本征函数相当于多个 SIMD 汇编语言指令，因此进行优化时，可能需要查看汇编语言输出，以确保即使使用内嵌 SIMD 汇编语言也不能进一步提高性能。

（9）到哪里去获取更多的信息

作者在少年时期，常常只需打一个电话，便可从公司免费地索要到价值数千美金的芯片、图书和软件。现在，读者仍可以这样做，但要找对人确实不容易。然而，Intel 在其网站上提供了大量的免费资料，读者可能很感兴趣。作者想提醒读者的是，可以通过 Intel 网站免费获得整套的奔腾 III/4 软件开发、编程和优化手册，这里说的不是 PDF 格式的文件，而是价值超过 250 美金的纸质图书，且送货上门。读者只需阅读这些图书，便可成为奔腾处理器优化大师。下面是可向 Intel 文献中心索要这些图书的网址：

http://developer.intel.com/design/pentium4/manuals/index2.htm

这包括下述用户手册：

- IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture。
- IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference。
- IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide。
- Intel Pentium 4 and Intel Xeon Processor Optimization Manual。

提示：这是世界上被保守得最严的秘密之一，除非您打算阅读这些免费图书，否则不要订阅。

什么也比不上免费的东西！Intel 网站中还有大量有关奔腾处理器优化的文章和白皮书。如果读者打算制作游戏，这些资料是必读的，花些时间阅读它们吧。我好高兴自己这样做了。

16.4.3 一个 SIMD 3D 向量类

现在，读者对 SIMD 理论有一定的了解，知道如何使用一些基本的数学指令，下面综合使用这些知识，创建一个支持 4D 向量的小型 C++类，并使用 SIMD 来执行计算。

这个类很原始，一点也谈不上完整，但对尝试使用 SIMD 进行编程而言已经很不错。这个类支持格式

为[x，y，z，w]的 4D 向量，因此其对象的数据格式与 SIMD 寄存器相同。作者在这个类中实现了加法、减法、点积、长度、打印、数组访问以及其他两个清理函数。另外，很多函数使用了条件编译，以支持 SIMD 汇编语言和 SIMD 本征函数。下面是用于控制条件编译的常量：

```
// 将一个常数设置为 1(另一个设置为 0)来指定使用哪种方式
#define SIMD_INTRINSIC   0
#define SIMD_ASM     1
```

下面是整个类的代码。其中包含非常详细的注释，请读者花些时间阅读它们，以理解每种 SIMD 运算。大部分运算都很容易理解，惟一理解起来可能有些困难的是点积，原因在于：计算点积时，并行地将两个向量的分量相乘很容易，但将 XMM 寄存器中的乘积相加比较困难，为得到最后的结果，需要使用指令 shufps（执行混合）和 addps。

```
// 这个新的向量类支持 SIMD SSE
// 这里提供了多种访问数据成员的方法
// 这为赋值、数据访问和使用本征库提供了透明的支持，而无需进行大量的强制数据类型转换
class C_VECTOR4D
{
public:

union
   {
   __declspec(align(16)) __m128 v;   // SIMD 数据类型存储方式
   float M[4];                  // 数组存储方式
   // 显式名称
   struct
      {
      float x,y,z,w;
      }; // end struct
   }; // end union

// 注意：上述 declspec 是多余的，因为__m128 要求编译器将数据与 16 字节边界对齐
// 因此只要共用体中有__m128，就没有必要使用 declspec
// 但使用它没有害处，定义局部变量和全局变量时
//使用 declspec(align(16))可确保它与 16 字节边界对齐

// 构造函数 ////////////////////////////////////////////////////////

C_VECTOR4D()
{
// 空构造函数
// 将向量初始化为 0.0.0.1
x=y=z=0; w=1.0;
} // end C_VECTOR4D

//////////////////////////////////////////////////////////////////////////

C_VECTOR4D(float _x, float _y, float _z, float _w = 1.0)
{
// 根据传入的参数值初始化向量
x = _x;
y = _y;
z = _z;
w = _w;
} // end C_VECTOR4D

// 成员函数 ////////////////////////////////////////////////////////

void init(float _x, float _y, float _z, float _w = 1.0)
{
// 根据传入的参数值初始化向量
x = _x;
y = _y;
z = _z;
w = _w;
```

```
} // end init

//////////////////////////////////////////////////////////////////////////

void zero(void)
{
// 将向量初始化为 0.0.0.1
x=y=z=0; w=1.0;

} // end zero

//////////////////////////////////////////////////////////////////////////

float length(void)
{
// 计算向量的长度
C_VECTOR4D vr = *this;

// set w=0
vr.w = 0;

// 纯汇编语言版本
#if (SIMD_ASM==1)

// 使用汇编语言来计算点积
// 根据点积来计算向量长度，因为 length = sqrt(v*v)
_asm
  {
  // 计算点积 this*this
  movaps xmm0, vr.v   // 将左操作数移到寄存器 xmm0 中
  mulps xmm0, xmm0    // 将两个操作数垂直相乘

  // 现在，xmm0 =
  // [ (v1.x * v2.x), (v1.y * v2.y), (v1.z * v2.z), (1*1) ]
  // 或者：假设 xmm0 = [x, y, z, 1] =
  // [ (v1.x * v2.x), (v1.y * v2.y), (v1.z * v2.z), (1*1) ]
  // 现在，需要将分量 x、y、z 相加，得到点积：
  // dp = x+ y +z = x1*x2 + y1*y2 + z1*z2

  // 开始计算
  // xmm0: = [x,y,z,1]
  // xmm1: = [?,?,?,?]
  movaps xmm1, xmm0 // 将结果复制到寄存器 xmm1 中
  // xmm0: = [x,y,z,1]
  // xmm1: = [x,y,z,1]

  shufps xmm1, xmm0, SIMD_SHUFFLE(0x01,0x00,0x03,0x02)
  // xmm0: = [x,y,z,1]
  // xmm1: = [z,1,x,y]

  addps xmm1, xmm0
  // xmm0: = [x  ,y  ,z  ,1]
  // xmm1: = [x+z,y+1,x+z,y+1]

  shufps xmm0, xmm1, SIMD_SHUFFLE(0x02,0x03,0x00,0x01)
  // xmm0: = [y  ,x  ,y+1,x+z]
  // xmm1: = [x+z,y+1,x+z,y+1]

  // 现在可以相加了
  addps xmm0, xmm1
  // xmm0: = [x+y+z,x+y+1,x+y+z+1,x+y+z+1]
  // xmm1: = [x+z  ,y+1  ,x+z    ,y+1]
  // xmm0.x 为点积
  // xmm0.z、xmm0.w 为点积+1

  // 现在 x 分量为点积，可以计算平方根了
  sqrtss xmm0, xmm0
```

```
    movaps vr, xmm0 // 保存结果

   } // end asm

#endif // end use inline asm version

// 使用本征函数
#if (SIMD_INTRINSIC==1)

#endif // end use intrinsic library version

// 返回结果
return(vr.x);

} // end length

// 运算符重载 ////////////////////////////////////////////////

float& operator[](int index)
{
// 返回数组中第 index 元素
return(M[index]);
} // end operator[]

//////////////////////////////////////////////////////////////////////////

C_VECTOR4D operator+(C_VECTOR4D &v)
{
// 将向量"this"和 v 相加

__declspec(align(16)) C_VECTOR4D vr; // 用于存储结果，与 16 字节边界对齐

// 使用汇编语言
#if (SIMD_ASM==1)

// 指出接下来要使用汇编语言
_asm
   {
   mov esi, this      // "this" 指向左操作数
   mov edi, v         // v 指向右操作数

   movaps xmm0, [esi]  // esi 指向第一个向量，将其移到寄存器 xmm0 中
   addps  xmm0, [edi]  // edi 指向第二个向量，将其加入到寄存器 xmm0 中

   movaps vr, xmm0     // 将结果存储到向量 vr 中

   } // end asm

#endif // end use inline asm version

// 使用本征函数?
#if (SIMD_INTRISIC==1)

vr.v = _mm_add_ps(this->v, v.v);

#endif // end use intrinsic library version

// 总是将 w 设置为 1
vr.w = 1.0;

// 返回结果
return(vr);

} // end operator+

//////////////////////////////////////////////////////////////////////////
```

```
C_VECTOR4D operator-(C_VECTOR4D &v)
{
// 将向量"this"与 v 相减

__declspec(align(16)) C_VECTOR4D vr; // 用于存储结果，与 16 字节边界对齐

// 使用汇编语言?
#if (SIMD_ASM==1)

// 指出接下来要使用汇编语言
_asm
  {
  mov esi, this       // "this" 指向左操作数
  mov edi, v          // v 指向右操作数

  movaps xmm0, [esi]  // esi 指向第一个向量，将其移到寄存器 xmm0 中
  subps  xmm0, [edi]  // edi 指向第二个向量，使用它去减寄存器 xmm0

  movaps vr, xmm0     // 将结果存储到向量 vr 中

  } // end asm

#endif // end use inline asm version

// 使用本征函数?
#if (SIMD_INTRISIC==1)

vr.v = _mm_sub_ps(this->v, v.v);

#endif // end use intrinsic library version

// 总是将 w 设置为 1
vr.w = 1.0;

// 返回结果
return(vr);

} // end operator-

////////////////////////////////////////////////////////////////////////////

float operator*(C_VECTOR4D &v)
{
// 运算符*表示计算点积
// 计算向量"this"和 v 的点积

__declspec(align(16)) C_VECTOR4D vr; // 用于存储结果，与 16 字节边界对齐

// 使用汇编语言?
#if (SIMD_ASM==1)

// 指出接下来要使用汇编语言
_asm
  {
  mov esi, this       // "this" 指向左操作数
  mov edi, v          // v 指向右操作数

  movaps xmm0, [esi]  // // 将左操作数移到寄存器 xmm0 中
  mulps xmm0,  [edi]  // 将两个操作数垂直相乘

  // 现在，xmm0  =
  // [ (v1.x * v2.x), (v1.y * v2.y), (v1.z * v2.z), (1*1) ]
  // 或者：假设 xmm0 = [x, y, z, 1] =
  // [ (v1.x * v2.x), (v1.y * v2.y), (v1.z * v2.z), (1*1) ]
  // 现在，需要将分量 x、y、z 相加，得到点积：
  // dp = x+ y +z = x1*x2 + y1*y2 + z1*z2
```

```
    // 开始计算
    // xmm0: = [x,y,z,1]
    // xmm1: = [?,?,?,?]
    movaps xmm1, xmm0 // 将结果复制到寄存器 xmm1 中
    // xmm0: = [x,y,z,1]
    // xmm1: = [x,y,z,1]

    shufps xmm1, xmm0, SIMD_SHUFFLE(0x01,0x00,0x03,0x02)
    // xmm0: = [x,y,z,1]
    // xmm1: = [z,1,x,y]

    addps xmm1, xmm0
    // xmm0: =[x  ,y  ,z  ,1]
    // xmm1: =[x+z,y+1,x+z,y+1]

    shufps xmm0, xmm1, SIMD_SHUFFLE(0x02,0x03,0x00,0x01)
    // xmm0: = [y  ,x  ,y+1,x+z]
    // xmm1: = [x+z,y+1,x+z,y+1]

    // 现在可以相加了
    addps xmm0, xmm1
    // xmm0: = [x+y+z,x+y+1,x+y+z+1,x+y+z+1]
    // xmm1: = [x+z  ,y+1  ,x+z    ,y+1]
    // xmm0.x 为点积
    // xmm0.z, xmm0.w 为点积+1
    movaps vr, xmm0
    } // end asm

#endif // end use inline asm version

// 使用本征函数?
#if (SIMD_INTRISIC==1)
vr.v = _mm_mul_ps(this->v, v.v);
return(vr.x + vr.y + vr.z);

#endif // end use intrinsic library version

// 返回结果
return(vr.x);

} // end operator*

//////////////////////////////////////////////////////////////////////////////

void print(void)
{
// 该成员函数打印向量
printf("\nv = [%f, %f, %f, %f]", this->x, this->y, this->z, this->w);
} // end print

//////////////////////////////////////////////////////////////////////////////

}; // end class C_VECTOR4D
```

稍后将介绍一些使用这个类的范例，但在此之前，读者可能注意到了，这里使用了本章前面有关数据移动的一节中介绍的宏 SIMD_SHUFFLE()。这个宏用于创建选择控制字，让能够使用指令 shufps 将源操作数中的 SIMD 字移到目标操作数中，将数据的次序排列好，以方便处理。

计算点积时，最后需要执行水平（horizontal）加法。使用并行 SIMD 乘法计算 u*v 得到的结果如下：

```
[ux*vx, uy*vy, uz*vz, 1*1]
```

问题是，要得到点积，需要计算（ux*vx + uy*vy + uz*uz）。这种计算无法使用常规 SIMD 指令来实现，因为没有水平加法指令。因此，需要复制乘积，对其进行移位，然后加入到一个累积器（accumulator）中，以得到下述最终结果：

```
(ux*vx + uy*vy + uz*vz)
```

下面来使用这个类。首先定义三个 SIMD 向量：

```
C_VECTOR4D v1(1,2,3), v2(4,5,6), result;
```

下面的代码将 v1 和 v2 相加：

```
result = v1+v2;
```

下面的代码打印 result：

```
result.print();
```

下面的代码计算并打印向量 result 的长度：

```
length = result.length();
cout << "\nlength";
```

最后，下面的代码计算向量 v1 和 v2 的点积：

```
float dp = v1*v2;
```

非常容易！DEMOII16_3.CPP|EXE 是一个使用这个类的演示程序，它包含整个类，并演示一些类似于上述范例的运算。和前一个演示程序一样，必须将该程序编译为控制台应用程序，同时您使用的操作系统必须支持 SSE，处理器必须是奔腾 III 或更好的。可尝试修改条件编译标记，为此修改本节前面定义的常量的值。

如果读者富有创造力，可尝试在类中添加矩阵乘法等想得到的任何功能。花些时间考虑每种运算的最快方式。与设计类一样，良好的 SIMD 编程要求您深思熟虑。如果只是蛮干，效果将不会比使用 C++代码好多少。

16.5 通用优化技巧

本节介绍一种对读者来说可能有用的优化技巧，它们有的是提示，有的是魔咒，有的是作者经验的结晶，有的来自本章后面列出的参考文献。

16.5.1 技巧 1：消除_ftol()

将 float 值赋给 int 变量时，编译器常常会调用内部函数_ftol()，这可能会降低性能。例如：

```
float f = 10.5;
int i = f;
```

为避免这种情况发生，可使用内嵌汇编指令和 FPU 指令，如 fistp/fst：

```
_asm
  {
  fld f;
  fistp i;
  }
```

或者使用编译器标记/QIfist 启用舍入模式，避免调用_ftol()。

然而，一定要注意这样做的副作用和降低精度，只针对模块启用舍入模式。_ftol() 调用是最大的性能瓶颈。您以为自己编写的代码是最优的，却不知道仅仅因为将 float 值赋给 int 变量，导致了调用函数_ftol()。

16.5.2 技巧 2：设置 FPU 控制字

浮点数处理器有一个控制字，它指定了舍入模式、精度和反向规格化（denormalization）。可以使用汇编语言来设置 FPU 控制字；然而，有一个能够为您完成这项工作的 C/C++运行时函数，它名为_control87()，如下所示：

```
unsigned int _control87( unsigned int control, unsigned int mask );
```

表 16.5 列出了一些用处较大的控制标记，您可以通过操纵它们来提高性能。

表 16.5　　_control87() 的参数

掩码（mask）	十六进制值	控制常量	十六进制值
_MCW_DN （反向规格化控制）	0x03000000		
		_DN_SAVE	0x00000000
		_DN_FLUSH	0x01000000
_MCW_RC（舍入模式控制）	0x00000300		
		_RC_CHOP	0x00000300
		_RC_UP	0x00000200
		_RC_DOWN	0x00000100
		_RC_NEAR	0x00000000
_MCW_PC（精度控制）	0x00030000		
		_PC_24（24 位）	0x00020000
		_PC_53（53 位）	0x00010000
		_PC_64（64 位）	0x00000000

例如，对非常小的数执行运算时，可能导致异常。然而，如果将反向规格化设置为刷新（flush），处理速度将提高。另外，在确保正常的情况下将计算精度降低，可提高计算速度。下面是设置这两个属性的代码：

```
// 设置为单精度
_control87( _PC_24, _MCW_PC );
// 设置为刷新模式
_control87( _DN_FLUSH, _MCW_DN );
```

16.5.3　技巧 3：快速将浮点变量设置为零

一般而言，整数和浮点数的二进制编码是完全不同的，也就是说，在二进制格式下，（int）1 和（float）1 是不相等的。然而，（int）0 和（float）0 确实相同，因此要快速清空大块内存区域，将其设置为浮点数 0，不应使用下述代码：

```
float x[1000];

for (int i=0; i < 1000; i++)
  x[i] = 0;
```

而应使用下面的代码：

```
memset(x, 0x0, sizeof(float)*1000);
```

另一种更好的方法是，使用内嵌汇编语言用一个 32 位字来填充内存，如下所示：

```
_asm
  {
  mov edi, x    ; edi 指向目标内存
  mov ecx, 1000/4 ; 要移动的 32 位字数
  mov eax, 0    ; 32 位数据
  rep stosd     ; 移动数据
  } // end asm
```

16.5.4　技巧 4：快速计算平方根

下面是一种快速计算平方根的算法，其误差小于 5%：

```
float Fast_SquareRT(float f)
{
float result; // 用于存储和返回结果
_asm
   {
  mov eax, f
  sub eax, 0x3f800000
  sar eax, 1
  add eax, 0x3f800000
  mov result, eax
  } // end asm

// 返回结果
return(result);

} // end Fast_SquareRT
```

16.5.5 技巧 5：分段线性反正切

本书前面介绍了使用查找表和其他技巧来计算反正切，然而这些技术都是针对特殊情形的。对于这个问题，有一种分段解决方案：对于任何(dx，dy)都能够计算出反正切，且误差非常小。

这种算法首先使用一个决策树来确定(dx，dy)位于哪个卦限（octant），然后使用相应的输出函数计算出一个[0，8]的值，您可以将结果缩放（重新映射）到任何范围，如 0～360 度。图 16.26 说明了这种算法。

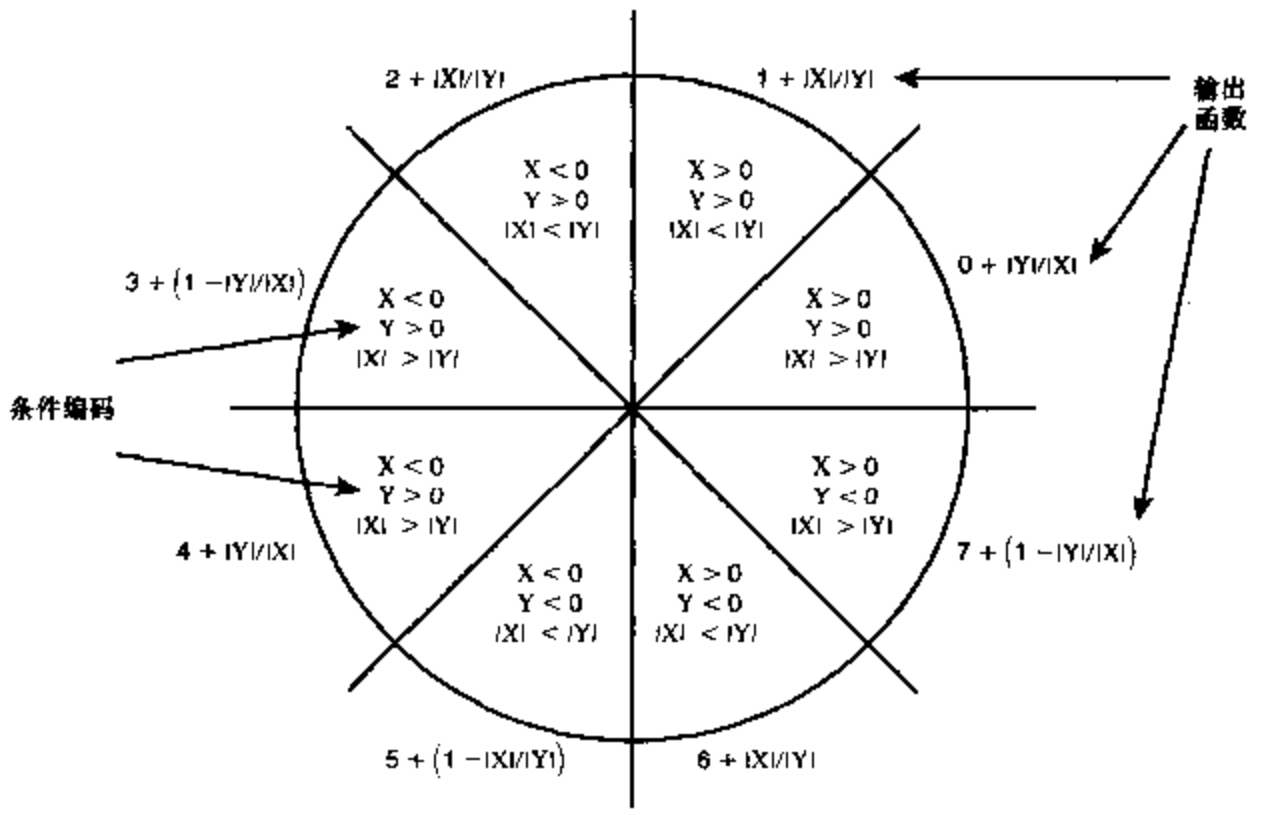


图 16.26 一种近似计算反正切的算法

16.5.6 技巧 6：指针递增运算

访问数组时，不要像下面这样使用指针递增运算：

```
int *p = base;

*p = 5;
p++;

*p = 6;
p++;

*p = 7;
```

而应这样做：

```
p[0] = 5;
p[1] = 6;
p[2] = 7;
```

与对指针执行递增运算，然后使用它来访问数组元素相比，使用索引表示法的速度更快。

16.5.7　技巧 7：尽可能将 if 语句放在循环外面

这是常识，但作者经常在代码中看到 if 语句被放在循环中，因此这里有必要指出这一点。假设有一个循环，在该循环中每隔 *n* 次迭代需要执行某种运算，因此您像下面这样做：

```
for (int i=0; i < 256; i++)
  {
  // 检测特殊情形
  if ((i % 64) == 0)
    {
    // 处理特殊情形的代码
    }
  else
    {
    // 处理常规情形的代码
    } // end else
  } // end for
```

这简直是灾难。为何不将上述循环分成 4 个，进而避免使用 if 语句呢：

```
// 处理特殊情形 i=128 的代码

// i 从 1~63 的循环
for (i = 1; i < 64; i++)
  // 处理常规情形的代码

// 处理特殊情形 i=64 的代码

// i 从 65~127 的循环
for (; i < 128; i++)
  // 处理常规情形的代码

// 处理特殊情形 i=128 的代码

// i 从 129~192 的循环
for (; i < 192; i++)
  // 处理常规情形的代码

// 处理特殊情形 i=192 的代码

// i 从 193~266 的循环
for (; i < 128; i++)
  // 处理常规情形的代码
```

16.5.8　技巧 8：支化（branching）流水线

奔腾 III/4 处理器的指令流水线非常长（最多可包含 20 条指令），当出现未预测到的分支时，将刷新整个流水线，因此应避免错误预测发生。另外，处理器执行代码时，通常会预先执行它认为接下来将出现的代码，因此出现预测错误时，不但需要刷新流水线，还会处理永远不会执行的代码。为避免预测错误，应使用大多数情况都能得到满足的 if 条件。

16.5.9　技巧 9：数据对齐

当数据与 16/32 字节边界对齐时，奔腾处理器的效率最高；因此，如果可能，应让所有的数据结构与这种边界对齐，并/或对数据结构进行填充，使其长度为 16/32 字节的倍数。要控制对齐，可以使用编译器选项，也可以使用方法__declspec（align（value））。

16.5.10 技巧 10：将所有简短函数都声明为内联的

对于任何被频繁调用且只有 10～20 行代码的函数，都应将其声明为内联的。要使函数成为内联的，必须将其从源文件移到头文件中，因为编译器不但需要函数头，还需要其源代码。例如，作者通过剖析后发现，向量点积函数和矩阵乘法函数被调用的次数比其他任何数学函数都多，因此将这些函数声明为内联的，结果性能提高了大约 3～5%，而这仅仅是通过增加一个关键字得到的。

16.5.11 参考文献

下面是本章使用的参考文献，它们也是获悉更多优化知识的好去处：

- The Graphics Gems Series (Academic Press)；
- IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture (Intel Press)；
- IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference (Intel Press)；
- IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide (Intel Press)；
- Intel Pentium 4 and Intel Xeon Processor Optimization Manual (Intel Press)；
- The Software Optimization Cookbook by Richard Gerber (Intel Press)。

16.6 总　　结

本章没有介绍核心代码的优化，而是将重点放在优化技术上，让读者知道如何使用一些可能从未使用过的技术和工具。对于本章讨论的技术，无论如何强调其重要性都不过分——它们将完全改变您有关下一代处理器和游戏应用程序的编程观点。使用秒表来优化代码的时代已一去不复返，您必须充分利用每一种现有的技术。